VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ
FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

ÚSTAV RADIOELEKTRONIKY
DEPARTMENT OF RADIO ELECTRONICS

MOBILNÍ APLIKACE PRO SPORT V PROSTŘEDÍ ANDROID

MOBILE ANDROID SPORT APPLICATION

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — Jan Maloušek
AUTHOR

VEDOUCÍ PRÁCE — Ing. Petr Kadlec, Ph.D.
SUPERVISOR

BRNO 2018

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

Bakalářská práce

bakalářský studijní obor **Elektronika a sdělovací technika**
Ústav radioelektroniky

Student: Jan Maloušek ***ID:*** 186445
Ročník: 3 ***Akademický rok:*** 2017/18

NÁZEV TÉMATU:

Mobilní aplikace pro sport v prostředí Android

POKYNY PRO VYPRACOVÁNÍ:

Seznamte se s prostředím Android Studio. Vytvořte aplikaci pro dotykové mobilní telefony, ve které půjde vytvářet, prohlížet a hlásit se na sportovní akce. V základním módu bude aplikace obsahovat alespoň 10 sportů a 10 sportovišť. V aplikaci půjde přidávat sporty a jednotlivá sportoviště. Dbejte zejména na to, aby ovládání aplikace bylo intuitivní (výběr pomocí komponent jako “Button, Slider, CalendarView” aj.).

Aplikaci doplňte tak, aby si v ní uživatel mohl vytvořit svůj vlastní účet (nejlépe již existující Google účet pomocí metody „sign in“). Dále zajistěte, aby si přihlášený uživatel mohl vybírat oblíbené sporty a ostatní uživatele. Doplňte aplikaci tak, aby aplikace nabízela akce z vybraného okolí, které zvolí uživatel. Aplikaci otestujte na skupině fiktivních uživatelů.

DOPORUČENÁ LITERATURA:

[1] SMITH, Dave. Android recipes: a problem-solution approach for Android 5.0. Fourth edition. ISBN9781484204764.

[2] BURD, Barry. Android application development all-in-one for dummies. 2nd edition. --For dummies. ISBN1118973801.

Termín zadání: 5. 2. 2018 ***Termín odevzdání:*** 24. 5. 2018

Vedoucí práce: Ing. Petr Kadlec, Ph.D.
Konzultant:

prof. Ing. Tomáš Kratochvíl, Ph.D.
předseda oborové rady

ABSTRAKT

Cílem této bakalářské práce je návrh mobilní aplikace pro OS Android sloužící k organizaci společných sportovních aktivit. Teoretická část popisuje samotný operační systém Android, tvorbu a základní komponenty aplikací a databázový systém Firebase. Druhá část práce se zabývá samotným návrhem aplikace včetně použitých řešení a algoritmů. Při návrhu bylo dbáno především na praktičnost a intuitivnost výsledné aplikace a na využívání doporučovaných programovacích postupů.

KLÍČOVÁ SLOVA

Android OS, mobilní aplikace, Firebase, Java, databáze, MVVM, RXJava2.

ABSTRACT

The aim of this Bachelor thesis is to design a mobile application for Android OS, which will be used to organize joint sports activities. Theoretical part of this document describes the Android operating system itself, creation and basic components of application and Firebase database system. Practical part of this thesis deals with designing the application itself, including used solutions and algorithms. Main focus of the design part was on functionality and intuitiveness and also on the use of the best practice programming techniques.

KEYWORDS

Android OS, Mobile application, Firebase, Java, database, MVVM, RXJava2.

Maloušek, J. *Mobilní aplikace pro sport v prostředí Android*. Brno: Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, Ústav radioelektroniky, 2018. 60 s., 0 s. příloh. Bakalářská práce. Vedoucí práce: Ing. Petr Kadlec, Ph.D.

PROHLÁŠENÍ

Prohlašuji, že svoji bakalářskou práci na téma Mobilní aplikace pro sport v prostředí Android jsem vypracoval samostatně pod vedením vedoucího bakalářské práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou všechny citovány v práci a uvedeny v seznamu literatury na konci práce.

Jako autor uvedené bakalářské práce dále prohlašuji, že v souvislosti s vytvořením této bakalářské práce jsem neporušil autorská práva třetích osob, zejména jsem nezasáhl nedovoleným způsobem do cizích autorských práv osobnostních a/nebo majetkových a jsem si plně vědom následků porušení ustanovení § 11 a následujících zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon), ve znění pozdějších předpisů, včetně možných trestněprávních důsledků vyplývajících z ustanovení části druhé, hlavy VI. díl 4 Trestního zákoníku č. 40/2009 Sb.

V Brně dne ....

(podpis autora)

PODĚKOVÁNÍ

Děkuji vedoucímu práce Ing. Petru Kadlecovi, Ph.D. za odbornou pomoc a cenné rady při zpracování této práce.

Dále bych chtěl poděkovat své rodině za veškerou podporu, Silvii Patzeltové za pomoc při návrhu loga a ikonky aplikace a všem testerům za jejich cenné připomínky.

OBSAH

SEZNAM OBRÁZKŮ

SEZNAM TABULEK

ÚVOD

Chytrá mobilní zařízení si zvláště v posledních letech vydobyla v lidském životě významné postavení. S technologickým vývojem se rozšiřují možnosti, jak mobilní zařízení využívat. Pokročilé chytré telefony tak v posledních letech zásadně změnily způsoby komunikace mezi lidmi a nahradily v této oblasti dosud dominující stolní počítače a notebooky.

Revoluce mobilního průmyslu se promítá do velké části lidských činností, včetně sportu. Cílem této práce je využít současných možností chytrých mobilních zařízení a vytvořit mobilní aplikaci, která by umožnila organizaci společných sportovních aktivit.

Při tvorbě této komunikační platformy jsou brány v potaz především intuitivnost, funkčnost a využívání ověřených programátorských praktik pro návrh aplikací. Grafický aspekt aplikace je navržen tak, aby byl v souladu s normami společnosti Google – Material Design [17].

V první části této práce je představen operační systém Android, pro který byla aplikace napsána. Nejdříve je představena historie samotného operačního systému, následuje krátké shrnutí jednotlivých verzí a poté je rozebrána jeho architektura.

Následující kapitola pojednává o platformě Firebase, včetně možnosti autentizace a cloudové databáze, na které je aplikace založena.

Třetí kapitola je zaměřena na vývoj aplikací pro operační systém Android. Je zde popsáno oficiální vývojové prostředí Android Studio a jeho nástroje, manifest aplikace a podrobně jsou zde rozebrány hlavní komponenty aplikací.

Čtvrtá kapitola pojednává o nejrozšířenějších architekturách využívaných při vývoji aplikací pro OS Android. V kapitole je nejprve rozebráno, proč vlastně architekturu potřebujeme a následně jsou popsány dva nejpoužívanější typy architektur MVP a MVVM.

V posledních dvou kapitolách je řešena praktická stránka této práce. Nejdříve je pojednáno o návrhu aplikace. Rozebrána je zde volba architektury, nutná povolení a další obecné charakteristiky, které byly při vývoji aplikace brány v úvahu. Druhá z praktických kapitol pak popisuje samotnou realizaci aplikace, zvolené algoritmy a řešení.

V závěru je shrnuto, čeho bylo při vývoji aplikace dosaženo a další možnosti potenciálního budoucího vývoje.

1 OPERAČNÍ SYSTÉM ANDROID

Operační systém Android je open-source platforma vyvíjená především pro nasazení v chytrých telefonech a tabletech. Samotný systém je založen na Linuxovém jádře různých verzí v závislosti na verzi OS (Operačního systému) Android.

1.1 Historie

Společnost Android, Inc. byla založena v říjnu roku 2003 Andym Rubinem, Chrisem Whitem, Nickem Searsem a Richem Minerem. Původním záměrem společnosti byl vývoj mobilních zařízení, která by dokázala reagovat na aktuální polohu a preference uživatele.

V srpnu roku 2005 firmu Android, Inc. odkoupila společnost Google, která v té době začala spolupracovat se softwarovými, hardwarovými a telekomunikačními společnostmi s cílem vstoupit na trh mobilních technologií. Ze spolupráce vzešlo sdružení OHA (Open Handset Alliance), což společnosti Google umožnilo vstoupit na trh s operačním systémem Android.

První beta-verze OS Android byly přístupné pouze společnosti Google a jejím partnerům v rámci sdružení OHA a byly pojmenovány po slavných robotech (Astro Boy, Bender a R2-D2). V srpnu roku 2008 přišla společnost Google s první komerční verzí Android 1.0. Společně s Android 1.1 to byly jediné dvě verze OS Android bez standardního pojmenování. Od Android 1.5 pak všechny verze tohoto operačního systému nesou standardní pojmenování podle cukrovinek srovnaných abecedně [2], [14].

1.2 Verze OS Android

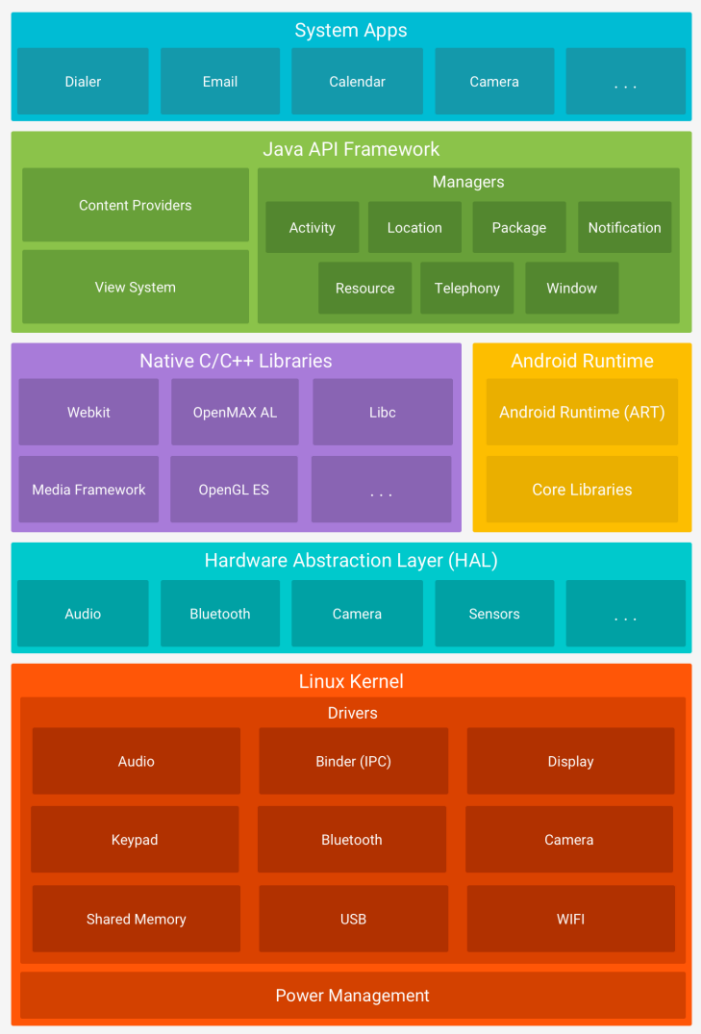
Operační Systém Android je pravidelně aktualizován a doplňován o novou funkcionalitu. Krátký přehled verzí je vidět na Tabulka 1.

Tabulka 1: Historie verzí OS Android [12], [13], [14]

Verze OS Android	Nové funkce v nich přidané
Android 1.5 Cupcake (04/2009)	Možnost nahrávání videí na YouTube a nahrávání fotografií do galerie Picasa. Podpora softwarových klávesnic třetích stran.
Android 1.6 Donut (09/2009)	Přepracovaný Android Market. Podpora různých rozlišení displeje. Upravené prostředí fotoaparátu.
Android 2.0-2.1 Eclair (10/2009)	Podpora přepisu mluveného slova. Navigace Google Maps.
Android 2.2 Froyo (05/2010)	Podpora hlasového ovládání. Podpora Wifi-hotspot. Zavedení JIT (Just-In-Time Compiler) – výrazné zvýšení výkonnosti.
Android 2.3 Gingerbread (10/2010)	Podpora NFC. Zlepšení UI. Optimalizované využívání baterie.
Android 3.0 Honeycomb (02/2011)	Podpora větších obrazovek (tablety). Notifikační lišta.
Android 4.0 Ice Cream Sandwich (10/2011)	Nové možnosti domovské obrazovky. Sledování využívání dat.
Android 4.1-4.3 Jelly Bean (06/2012)	Podpora více uživatelských účtů. Zlepšení běhu animací a zrychlení odezvy systému.
Android 4.4 KitKat (09/2013)	Optimalizace systému. Úprava UI.
Android 5.0 Lollipop (11/2014)	Podpora 64-bitových procesorů. Unifikace UI za pomocí Material Design. Nahrazení virtuálního stroje Dalvik za ART.
Android 6.0 Marshmallow (10/2015)	Lepší správa povolení aplikacím. Optimalizace systému pro vyšší výdrž baterie.
Android 7.0 Nugat (10/2016)	Podpora rozdělení obrazovky (multitasking). Podpora Vulcan Api pro 3D grafiku.
Android 8.0 Oreo (08/2017)	Lepší správa hesel. Lepší správa baterie. Lepší správa notifikací.

1.3 Architektura

Architektura operačního systému Android je rozdělena do pěti vrstev - Linuxové jádro, Hardware Abstraction Layer (HAL), knihovny + Android Runtime, Application framework a aplikace. Přestože každá vrstva má na starosti určité operace, v realitě spolu jednotlivé vrstvy spolupracují a nejsou tak od sebe striktně odděleny. Celková struktura systému je vidět na Obr. 1.



Obr. 1: Architektura OS Android [1].

1.3.1 Linuxové jádro

Jedná se o nejnižší vrstvu operačního systému Android, která představuje určitou abstrakci mezi hardwarem daného zařízení a softwarem ve vyšších vrstvách operačního systému Android. Linuxové jádro tak například zajišťuje správu pamětí, sítí a procesů, zabezpečení systému atd. První verze OS Android byly založené na Linuxovém jádře verze 2.6.x, současná verze OS Android - Android 8.0 Oreo (k datu 20. 5. 2018) - využívá Linuxové jádro verze 4.4 [8], [9], [11].

1.3.2 Hardware Abstraction Layer

Vrstva HAL (Hardware Abstraction Layer) obsahuje knihovní moduly poskytující standardizovaná rozhraní pro hardwarová zařízení (kamera, bluetooth modul, …). Využívání standardizovaných hardwarových rozhraní umožňuje výrobcům modifikovat hardwarové vybavení zařízení, aniž by tím ovlivnili vyšší vrstvy systému. Jednotliví výrobci mohou vrstvu HAL rozšířit o vlastní rozhraní, avšak za předpokladu, že splní potřebné specifikace a požadavky.

1.3.3 Android Runtime

Zdrojový kód aplikací je nejprve zkompilován do tzv. *bytecode*, a ten je následně interpretován virtuálním strojem [10]. Starší verze OS Android až po verzi Android 4.4 Kitkat využívaly vlastní virtuální stroj od společnosti Google se jménem Dalvik. Ten obsahoval funkcionalitu JIT (Just In Time) kompilátoru, kdy *bytecode* aplikace byl kompilován za běhu. Od Androidu verze 5.0 Lollipop byl virtuální stroj Dalvik nahrazen běhovým prostředím ART (Android Runtime).

ART využívá na rozdíl od virtuálního stroje Dalvik primárně AOT (Ahead-Of-Time) kompilaci, tedy *bytecode* aplikace je zkompilován kompletně již při instalaci. Výhodou je rychlejší běh aplikací a prodloužení životnosti baterie, nevýhodou je pak delší doba instalace aplikace (aplikace se musí po instalaci navíc zkompilovat). Mezi další výhody ART patří také vylepšený Garbage Colector, nebo zlepšené možnosti debuggingu při vývoji aplikací.

Od Androidu verze 7.0 Nugat byl do ART přidán JIT kompilátor, který napomáhá k optimalizaci výkonnosti aplikací při jejich běhu, a doplňuje tak AOT kompilátor.

Součástí vrstvy Android Runtime jsou kromě samotného virtuálního stroje ART také základní knihovny převzaté z jazyka Java [1].

1.3.4 Knihovny

Knihovní vrstva operačního systému Android obsahuje nejrůznější knihovny s celou řadou rozhraní API využitelných při vývoji aplikací. Dále se zde nacházejí nativní C/C++ knihovny jako je například SGL, OpenGL, SQLite a další, které slouží k vykreslování 2D/3D grafiky, k práci s relačními databázemi atd. Funkce těchto knihoven jsou vývojářům aplikací poskytovány prostřednictvím Android Application Framework [1], [11].

1.3.5 Application Framework

Application Framework je velmi důležitá vrstva především pro vývojáře aplikací, neboť poskytuje mnoho vysokoúrovňových služeb, které jednotlivé aplikace mohou za svého běhu využívat. Mezi nejdůležitější služby Application Framework patří:

Activity Manager - zajišťuje životní cyklus aplikací a jejich aktivit.

Content Providers - umožňuje přístup k datům a sdílení dat mezi aplikacemi.

Resource Manager – poskytuje přístup k "nekódovým" zdrojům aplikace. Mezi tyto zdroje patří například lokalizované textové řetězce, obrázky, ikony, grafika, layout

soubory aj.

Notification Manager – umožňuje aplikacím zobrazovat ve stavovém řádku vlastní upozornění.

Sada komponent View - jedná se o službu poskytující základní prvky uživatelského rozhraní jako jsou textová pole, tlačítka, seznamy, checkboxy atd. [8].

1.3.6 Aplikace

Nejvyšší vrstvu architektury OS Android představují samotné aplikace. Může se jednat o aplikace předinstalované výrobcem zařízení, stažené prostřednictvím oficiálního obchodu Google Play, nebo o aplikace stažené z jiných zdrojů.

Až na určité výjimky (například aplikace nastavení) nemají předinstalované aplikace žádný speciální status. Defaultní aplikací například pro webový prohlížeč, klávesnicí a jiné služby se tak může stát jakákoliv aplikace, kterou uživatel nainstaluje [1].

2 FIREBASE

Firebase je multi-platformní systém spadající pod společnost Google, který poskytuje služby spojené s cloudovou databází a úložištěm. Hlavní výhodou Firebase je zajištění veškerých serverových řešení, při poskytnutí jednoduchých API pro vývojáře aplikací třetích stran.

Firebase nabízí klientské knihovny pro spravování databáze. Knihovny jsou napsány pro celou řadu programovacích jazyků, mezi kterými je Java, SWIFT, C++, PYTHON, JavaScript a další.

Hlavní službou společnosti Firebase je realtime noSQL databáze. Kromě ní však Firebase nabízí několik dalších služeb jako je autentifikace, cloudové úložiště, Crash Reporting aj. [15].

2.1 Autentifikace

Autentifikace uživatelů na Firebase může probíhat za využití několika metod. K dispozici jsou API pro přihlášení přes účty Googlu, Facebooku, Twitteru či GitHubu, případně je možné přihlašování provádět prostřednictvím e-mailu nebo telefonního čísla. Kromě klasických účtů podporuje Firebase rovněž anonymní uživatele, kteří se mohou do konkrétní aplikace anonymně přihlásit, a časem, pokud se tak rozhodnou, upgradovat anonymní účet na klasický, aniž by ztratili již zadaná data.

Při přihlášení jsou přihlašovací údaje nebo token (v případě přihlašování přes účty třetích stran) následně odeslány Firebase k ověření a klientská strana (vlastní aplikace) je poté informována o výsledku přihlašovacího procesu [16].

2.2 Realtime databáze

Firebase Realtime databáze je cloudová noSQL databáze umožňující synchronizaci dat v reálném čase napříč všemi uživateli. Data v databázi jsou uložena ve formátu JSON. Data jsou v databázi uložena do stromovité struktury s maximálně 32 úrovněmi. Přístup k jednotlivým částem databáze lze pak komplexně nastavit za pomocí autentifikace, kromě toho lze stanovit i validační skripty, pomocí nichž lze provést validaci vkládaných dat [15].

Databáze v aplikaci může zůstat funkční i v offline režimu, kdy jsou vložená data uložena v místním úložišti a po obnovení připojení se synchronizují s cloudovou databází [16].

3 TVORBA APLIKACÍ PRO OS ANDROID

3.1 Android Studio

Android Studio je oficiální programovací prostředí (IDE - Integrated Development Environment) pro Android aplikace přímo od společnosti Google. První stabilní verze vývojového prostředí Android Studio byla vydána v prosinci roku 2014. Před příchodem Android Studia bylo pro vývoj aplikací pro Android OS oficiálně používáno vývojové prostředí Eclipse s pluginem Android Developer Tools.

3.1.1 Nástroje

Android Studio obsahuje několik pokročilých vývojářských funkcí, které umožňují zefektivnit psaní aplikací a zlepšit kvalitu kódu.

Gradle - nástroj Gradle slouží k sestavení aplikačního souboru typu APK (Android Application Package). Při kompilování Gradle nástroj konvertuje zdrojové kódy do souborů typu DEX (Dalvik Executable), které umí zpracovávat virtuální stroj ART (případně Dalvik) v OS Android. Následně jsou soubory typu DEX a zkompilované Resources spojeny do jednoho aplikačního balíčku typu APK, a poté je soubor APK podepsán unikátním klíčem. Nakonec nástroj ZipAlign optimalizuje celý soubor tak, aby využíval na koncových zařízeních co nejméně paměti.

V Gradle je rovněž možné definovat dependencies (využívané knihovny), verzi aplikace, TargetSDK, CompileSDK, a MinimumSDK a další podrobnosti pro sestavení.

Debugger - pomocí Debuggeru lze procházet kód příkaz po příkazu, vyčítat aktuální hodnoty proměnných a případně zjišťovat, které částí programu způsobují pád aplikace.

Lint - nástroj Lint kontroluje kód a upozorní na části, které mohou dělat problémy. Nastroj tak upozorňuje například na využívání zastaralých metod, na rizikové části programu, na proměnné, kterým lze nastavit privátní atribut na místo veřejného atd. Každý zaznamenaný problém je zvýrazněn s vysvětlujícím textem.

Emulátor - Android Studio obsahuje rovněž emulátor zařízení s operačním systémem Android. Emulátor dokáže simulovat mnoho druhů telefonů, tabletů a dalších zařízení včetně jejich senzorů, připojení k sítí atd.

Našeptávač - stejně jako mnoho dalších IDE (Visual Studio, Eclipse…) také Android Studio obsahuje inteligentní našeptávač, který dokáže urychlit a usnadnit psaní kódu automatickými návrhy na doplnění rozepsaných příkazů.

Layout editor - umožňuje navrhovat UI (User Interface) metodou "drag and drop", případně textově v souborech typu xml. Editor obsahuje mnoho předpracovaných komponent včetně tlačítek, textových polí, seznamů, atd. U jednotlivých komponent UI se mimo jiných parametrů může nastavit také ID, pomocí kterého se následně komponenty přiřadí ve zdrojovém kódu k listenerům [1], [3].

3.2 Android manifest

Každá aplikace pro OS Android musí obsahovat soubor AndroidManifest.xml, který poskytuje všechny důležité informace, které systém musí mít ještě před tím, než se samotná aplikace spustí.

Součástí manifestu je popis aplikace včetně package name (unikátní identifikátor aplikace), ikonky a všech komponent (aktivity, service, broadcastReceiver, contentProvider atd.). Pomocí povolování a zakazování aktivit lze snadno řídit procesy, které se spustí jen při prvním spuštěním aplikace, což je vhodné například pro nastavení profilu, nebo pro naznačení ovládání aplikace uživateli.

Další důležitou součástí manifestu jsou žádosti o různá povolení (k přístupu k internetu, k zápisu do interního úložiště, ke čtení sms, atd.), které uživatel musí všechny schválit před nainstalováním aplikace (v případě Androidu do verze 5.1), nebo o které aplikace musí zažádat za běhu (od Androidu verze 6.0).

Manifest může rovněž obsahovat informace o tom, zdali může být aplikace instalována na SD kartu, jaký hardware aplikace využívá (díky čemuž pak Obchod Google Play nebude nabízet aplikaci zařízením, které konkrétní hardware neobsahuje), jaké typy obrazovek jsou podporovány (opět pro filtraci, kterým zařízením bude aplikace nabízena) atd. [1], [7].

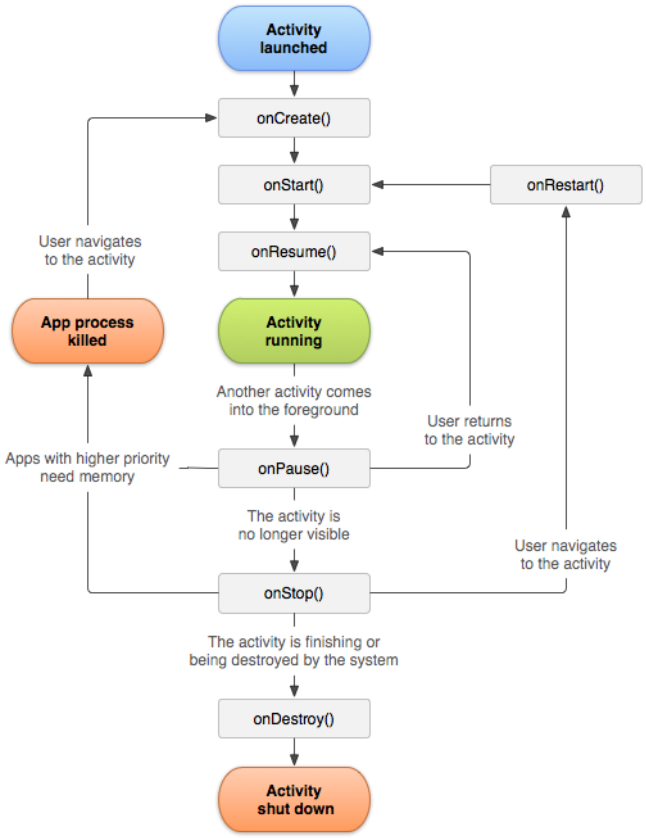
3.3 Komponenty aplikací

3.3.1 Aktivita

Aktivita je jedním ze základních bloků aplikace. Jedná se o komponentu, zajišťující uživatelské rozhraní v podobě jedné “obrazovky” aplikace. Všechny aplikace obsahují minimálně jednu, většinou však více aktivit, z nichž jedna musí být v manifestu označena jako „MAIN“. Tato aktivita je pak inicializována společně se spuštěním aplikace.

Životní cyklus aktivity

Každá aktivita má svůj vlastní životní cyklus, v rámci něhož přechází mezi různými stavy a systém při tom spouští určité metody, které mají vývojáři aplikací k dispozici. Životní cyklus aktivit se řídí podle následujícího diagramu, viz Obr. 2.



Obr. 2: Životní cyklus aktivit [1].

Aktivita se může nacházet v celkem čtyřech základních stavech:

1. ***Running*** - aktivita poté, co byla úspěšně spuštěna nebo obnovena, běží na popředí.

2. ***Paused*** - aktivita je stále vidět, ale je překryta například dialogovým oknem. V tomto stavu si zachovává veškeré své členy, avšak uživatel s ní nemůže nijak interagovat.

3. ***Stopped*** - aktivita je překryta jinou aktivitou a není tak již vidět. Samotná aktivita ale zatím nebyla zničena. Uživatel se k ní může vrátit, pokud nebude zničena operačním systémem například v případě nedostatku paměti.

4. ***Destroyed*** - aktivita byla zničena operačním systémem Android a pokud se k ní uživatel bude chtít vrátit, musí být kompletně restartována a obnovena.

Metody životního cyklu aktivity:

onCreate() – v této funkci by měly být provedeny všechny kroky nutné pro vytvoření dané aktivity. V *onCreate()* se obyčejně nastavuje layout aktivity, inicializují se

globální proměnné apod. Metoda přijímá jako parametr objekt typu *Bundle* s názvem *savedInstanceState*, pomocí kterého může být aktivita opětovně inicializována do stavu, ve kterém byla před svým zničením.

onStart() – metoda *onStart()* proběhne ve chvíli, kdy se aktivita spouští. Po proběhnutí této metody je aktivita uživateli již viditelná, avšak uživatel s ní zatím ještě nemůže interagovat. V metodě *onStart()* jsou inicializovány objekty, které jsou následně zničeny v metodě *onStop()*, příkladem těchto objektů mohou být například listenery.

onResume() – poté, co se aktivita přesune do popředí, se spustí metoda *onResume()*. V této chvíli už aktivita přijímá vstup od uživatele. Metoda *onResume()* je volána těsně po metodě *onStart()*, případně poté, co je aktivita obnovena ze stavu *Paused*. Metoda *onResume()* se používá k inicializování objektů, které jsou následně zničeny v metodě *onPause()*. Příkladem využití metody *onResume()* může být například registrace BroadcastReceiveru.

onPause() – metodu *onPause()* systém volá pokaždé, když má být aktivita přesunuta do pozadí. Funkce se tak využívá k uvolnění systémových zdrojů, tedy například ke zrušení registrace BroadcastReceiveru. Ve funkci *onPause()* musí být ukončeny také všechny děje, které nemají pokračovat v případě, že aktivita už není v popředí. V *onPause()* by se neměly vykonávat žádné časově náročné úlohy, neboť by se nemusely stihnout dokončit.

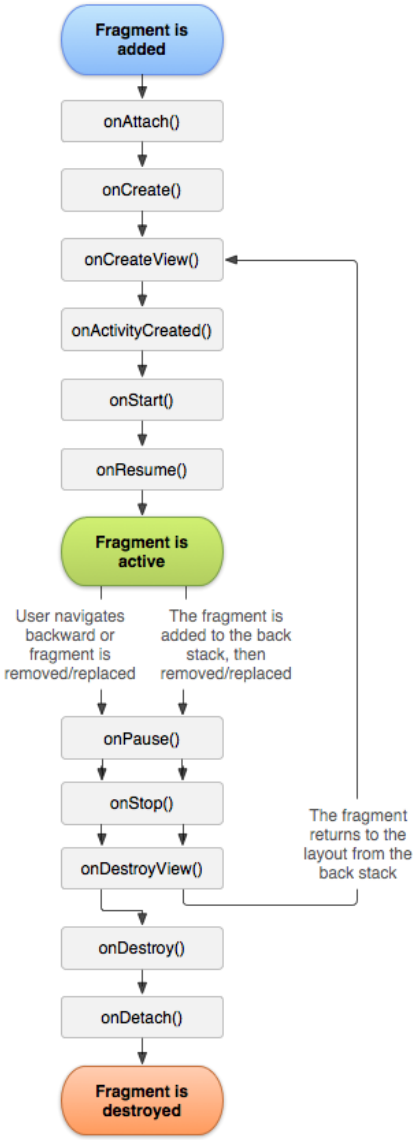
onStop() – funkce *onStop()* je vyvolána poté, co aktivita už není viditelná uživateli. Příkladem může být spuštění nové aktivity, která překryje stávající. Metoda by měla uvolnit téměř všechny zbývající zdroje, které aktivita nepotřebuje v době, když uživatel aktivitu nevyužívá. V metodě *onStop()* by tak měly být uvolněny zdroje inicializované v metodě *onStart()*. Funkce *onStop()* je rovněž vhodná k provedení všech časově náročných úloh související s ukončením aktivity, například uložení záznamů do databáze.

onRestart() – funkce *onRestart()* je vyvolávána v případě, že systém obnovil aktivitu ze stavu *Stopped*.

onDestroy() – jedná se o poslední metodu aktivity, která je provedena před tím, než systém aktivitu zničí [1], [2], [3], [6].

3.3.2 Fragment

Fragmenty jsou ve své podstatě modulární sekce v aktivitě s vlastním životním cyklem, viz Obr. 3. Jednou z výhod fragmentů je, že na rozdíl od aktivit lze v aplikaci zobrazit více fragmentů najednou. Dalším z hlavních rysů fragmentů je, že mohou být znovu využity v celé aplikaci. Fragment může mít stejně jako aktivita svůj vlastní přidružený layout soubor typu XML, ve kterém je definováno jeho vlastní UI, případně lze fragmenty využít i jako neviditelné pracovníky aktivit.



Obr. 3: Životní cyklus fragmentu [1].

Protože fragmenty jsou součástí aktivit, je nutné mezi aktivitami a fragmenty zabezpečit komunikaci. Komunikace z fragmentu do mateřské aktivity probíhá za pomocí *Interface*, který musí mateřská aktivita implementovat. Komunikace opačným směrem, tedy od mateřské aktivity k fragmentu, se uskutečňuje přímým voláním metod fragmentu z mateřské aktivity. Dalším možným způsobem jak zabezpečit vzájemnou komunikaci aktivit a fragmentů jsou tzv. *LocalBroadcast* [1], [3].

3.3.3 Intent

Intenty jsou objekty abstraktně popisující operace, které mají být provedeny. Využívány jsou při zahajování nových aktivit, odesílání *Broadcastů* apod. Samotný *Intent* v sobě může nést i data. *Intenty* se dělí na dvě skupiny, a to na implicitní a explicitní.

Explicitní *Intenty* přesně specifikují komponenty, které mají být spuštěny, prostřednictvím položky „*Component name*“. Tyto *Intenty* se využívají například při

zahájení nové aktivity.

Implicitní *Intenty* přesně nespecifikují komponenty, které mají být spuštěny, ale musí obsahovat dostatek informací k tomu, aby systém mohl sám determinovat kterou komponentu spustit. Příkladem může být funkce v aplikaci, která prostřednictvím emailu poskytne uživatelskou zpětnou vazbu vývojáři. Aplikace tak vyšle pouze implicitní *Intent*, a systém Android sám rozhodne, který emailový klient bude spuštěn [1], [5].

3.3.4 Service

Service je komponenta, která provádí dlouhotrvající operace na pozadí. *Service* bývají využity ke stahování dat z internetu, přehrávání hudby na pozadí, v antivirových aplikacích apod. Po svém spuštění může *Service* pokračovat ve své práci i poté, co uživatel přepne na jinou aplikaci.

Ukončení *Service* může nastat několika způsoby, prvním z nich je, že *Service* dokončí vlastní práci a zavolá metodu *stopSelf()*. Druhou možnosti je zastavení *Service* jinou komponentou, která zavolá funkci *stopService()*. Poslední třetí možností je ukončení *Service* systémem Android v případě, že potřebuje uvolnit prostředky k vykonání činností s vyšší prioritou [1], [3], [4].

3.3.5 Broadcast

Broadcasty jsou ve své podstatě zprávy vysílané napříč systémem Android, případně mohou být omezené pouze na jednu aplikaci. Komponenty jednotlivých aplikací pak mohou přijímat specifické *Broadcasty* a reagovat na ně.

Ve chvíli, kdy systém nebo nějaká aplikace vyšle *Broadcast*, OS Android automaticky přesměruje tyto *Broadcasty* aplikacím, které na ně mají zaregistrovaný *BroadcastReceiver*. Součástí *Broadcastu* mohou být i data uložená v *Intentu*, například ve formě objektu typu *Bundle*.

V případě, že není nutné vysílat *Broadcast* mimo aplikaci, je vhodné využít *LocalBroadcastManageru*. Tato implementace je nejen efektivnější, ale hlavně bezpečnější, neboť vyslaná data nemůže zachytit žádná jiná aplikace [1], [4].

3.3.6 ContentProvider

ContentProvider je komponenta aplikace, která je využívána ke čtení a zápisu dat do úložišť, databází apod. *ContentProvidery* mohou rovněž hlídat změny v datech aplikace a automaticky upozorňovat na změny prostřednictvím *Listenerů*. *ContentProvidery* zároveň poskytují mechanizmy pro definování zabezpečení dat [1], [3].

3.4 Resources

Resources je statická třída v aplikaci, která slouží k přístupu k “nekódovým“ zdrojům aplikace (obrázky, textové řetězce, ikonky, barvy, apod.). Pomocí *Resources* lze snadno vytvořit jazykovou lokalizaci aplikace, kdy všechny textové řetězce se uloží v různých jazykových variacích, a systém Android následně automaticky podle nastaveného

jazyku určí, kterou jazykovou variaci využít. Podobně je to i s obrázky či ikonkami, kdy v *Resources* lze uložit jednotlivé obrazové soubory v různých rozlišeních a systém Android pak bude v aplikaci volit využití obrazových zdrojů v závislosti na velikosti a rozlišení displeje [1], [3].

3.5 Room

Knihovna *Room* od společnosti Google poskytuje abstrakci nad SQLite databází v aplikacích pro OS Android. Jejím cílem je odstranění velkého množství kódu (tzv. *boilerplate code*), které bylo nutné napsat k vytvoření SQLite databází s původní doporučovanou knihovnou *SQLiteOpenHelper*.

Knihovnu *Room* a SQLite databázi je vhodné použít kdykoliv je potřeba v aplikaci uložit nějaká rozsáhlejší data, která si přejeme zachovat i po vypnutí aplikace. Databáze může vracet data nejen tradičních typů jako *List*, *String* atd., ale i pozorovatelná data jako *LiveData* nebo *Flowable*. *Room* se skládá ze tří hlavních částí:

- **Entita**

Entita představuje jeden řádek v databázi, přičemž jedno pole z těchto entit slouží jako primární klíč. Každý typ entity je následně ukládán do vlastní SQL tabulky.

- **DAO (Data Access Object)**

K samotným datům v databázi se přistupuje pomocí *DAO*. Jedná se o interface s metodami pro čtení, mazání a zápis dat, jejichž chování je určeno přiloženými SQL příkazy.

- **Database**

Jedná se o objekt, který v sobě drží seznam *Entit* a *DAO*. Pomocí tohoto objektu je také možné definovat chování databáze, například povolit přístup k databázi v hlavním vláknu (pouze pro testovací účely), definovat migraci dat při změně entit a podobně [1], [18], [19], [20].

3.6 Reaktivní programování a RxJava2

RxJava2 je knihovna od společnosti ReactiveX, která využívá principu reaktivního programování. Knihovna tak tvoří abstrakci nad prací s vlákny, kdy je potřeba časově náročné úkony, jako je například přístup k databázi nebo složité výpočty, přesunout pryč z hlavního vlákna, které je zodpovědné za UI (user interface), s cílem zachovat responzivnost a plynulost aplikací.

Knihovna je založena na vzoru *Observable-Observer*. V případě, že potřebujeme vykonat nějakou časově náročnou úlohu, vytvoříme objekt typu *Observable*, do kterého zapouzdříme potřebnou funkcionalitu, kterou by bylo vhodné vykonávat ve vlákně na pozadí. *Observables* tedy představují datový tok, kde jsou data zpracovávána a následně emitována. Objekt typu *Observer* přihlásíme k *Observables* pomocí metody *subsribe()*. Jakmile je práce dokončena, objekt *Observable* pošle, neboli emituje, výsledek všem *Observerům*, které jej dále zpracují (zobrazí data uživateli apod.).

V knihovně RxJava2 existuje několik typů *Observables*:

Completable: *Observable*, který pouze vykoná požadovanou práci, po jejímž dokončení notifikuje *Observery* pomocí metody *onComplete()*, která nemá žádné parametry. *Completables* jsou tedy vhodné na vykonávání práce bez návratové hodnoty.

Single: jedná se o podobný typ jako *Completable*, avšak s tím rozdílem, že po dokončení práce vyžadujeme nějakou návratovou hodnotu.

Observable: v případě, že během práce potřebujeme notifikovat hlavní vlákno, například o jejím průběhu, je vhodné použít *Observable*, který během práce může posílat mezivýsledky *Observerům* pomocí metody *onNext()* a po dokončení poslat konečný výsledek pomocí metody *onComplete()*.

Flowable: ve své podstatě je *Flowable* podobný *Observable*, oproti *Observable* ale implementuje takzvanou *BackPressure* strategii. Tato strategie popisuje, jak se má *Flowable* zachovat v případě, že emituje výsledky rychleji, než je zvládá *Observer* zpracovávat. Z toho vyplývá, že *Flowable* by se měl použít tam, kde očekáváme rychlé emitování velkého množství výsledků [21], [22], [23].

3.7 Dependency Injection a Dagger2

Cílem návrhového vzoru *Dependency Injection (DI)* je snížení závislostí mezi částmi programu, kdy namísto, aby byly všechny závislosti daného objektu nastaveny jiným objektem, který jej vytvořil, jsou závislosti do objektu „injektovány“ externě. Výsledkem je přehlednější a lépe testovatelný program.

Jedna z často používaných knihoven pro *DI* určených pro Android je *Dagger2*. *Dagger2* automaticky generuje kód potřebný k *DI*, čímž se opět sníží množství *boilerplate kódu*. Při využívání *Dagger2* je nejprve potřeba definovat *Komponenty* a *Moduly*. *Moduly* popisují potřebné závislosti, přičemž *Komponenty* definují, kam mohou být tyto závislosti injektovány. K injektování závislosti do konkrétního objektu se používá anotace *@inject*, přičemž injektovat lze v konstruktoru, metodě, nebo přímo do pole [24], [25], [26].

4 ARCHITEKTURA APLIKACE

S množstvím funkcí aplikace roste i složitost a komplexnost zdrojového kódu. V určitém bodě se tak zdrojový kód stává velmi špatně čitelný a nepřehledný. Důsledkem toho je menší spolehlivost kódu, případné chyby se hůře opravují a přidávání nové funkcionality je velmi náročné. Z těchto důvodu je při návrhu aplikace nutné přemýšlet i nad architekturou, a logicky aplikaci rozdělit do jednotlivých modulů, které budou ideálně následovat tzv. *Single responsibility patern* – tedy každý modul aplikace má na starost jednu určitou úlohu.

Jednotlivé moduly je vhodné následně rozdělit do určitých vrstev, které mezi sebou budou komunikovat. Při vývoji aplikací pro OS Android se často využívají tři vrstvy, a to vrstva uživatelského rozhraní, vrstva logiky a vrstva datová. Vrstvy lze uspořádat různými způsoby, přičemž při vývoji aplikací se v současné době často využívají architektury *Model-View-Presenter* a *Model-View-ViewModel*.

4.1 Model-View-Presenter (MVP)

Aplikace je rozdělena na tři vrstvy s názvy *Model*, *View* a *Presenter*.

Vrstva View představuje uživatelské rozhraní zastoupené v aplikacích Android například Aktivitami. Požadavkem této vrstvy je co největší jednoduchost. Jedinou úlohou tříd ve vrstvě *View* je zobrazení dat a předání uživatelských interakcí vrstvě *Presenter*.

Vrstva Presenter slouží jako prostředník mezi vrstvami *Model* a *View*. *Presenter* má přístup jak do Modelu tak do *View*, přičemž obě vrstvy ve své podstatě ovládá. V *Presenteru* by se rovněž měla realizovat veškerá logika aplikace. Jednou ze zásad *Presenteru* je, že by měl být nezávislý na jakýchkoliv knihovnách Androidu. Vztah mezi *View* a *Presenterem* by měl být jeden na jednoho, přičemž k jednomu *Presenteru* může být připojeno několik *Modelů*.

Vrstva Model obsahuje třídy zodpovědné za přístup k datům, ať už se jedná o vnitřní úložiště nebo cloudovou databázi. K vrstvě *Model* má přístup pouze vrstva *Presenter*, která si vyžádá v *Modelu* data a Model vrstvě *Presenter* následně data doručí.

Komunikace mezi vrstvami může probíhat přímo voláním veřejných metod jednotlivých vrstev, ale z důvodu řádného oddělení vrstev a snazšímu testování je lepší zajistit komunikaci prostřednictvím takzvaných kontraktů, neboli interfaců, které v sobě obsahují pouze metody, které mohou být přístupné z jiné vrstvy [27], [28].

4.2 Model-View-ViewModel (MVVM)

Aplikace je opět rozdělena na tři vrstvy, tentokrát pojmenované *Model*, *View* a *ViewModel*.

Vrstva View má za úkol, stejně jako tomu bylo v architektuře MVP, zobrazovat uživatelská data a přeposílat uživatelské pokyny třídě *ViewModel*. Na rozdíl od MVP ale *ViewModel* vrstvu *View* neovládá, vrstva *View* si musí o zobrazitelná data požádat *ViewModel* prostřednictvím *Getteru*, využitím takzvaného *DataBinding*, případně využít observovatelných dat ve *ViewModelu*, jako jsou například *LiveData*.

ViewModel opět figuruje jako prostředník mezi vrstvami *View* a Model. Komunikace mezi vrstvami *ViewModel* a Model probíhá obdobně jako u architektury MVP prostřednictvím interfaců neboli kontraktů. *ViewModel* ale tentokrát nemá žádnou referenci na vrstvu *View*, nemůže k ní tedy nijak přistupovat nebo ji jakkoliv přímo ovládat. Pro vrstvu *View* tak pouze připravuje data, například v podobě *LiveData*, ke kterým může vrstva *View* zaregistrovat *Observera*.

Vrstva Model je obdobná jako u architektury MVP, opět slouží k přístupu k datům v místním i cloudovém úložišti a předání těchto dat *ViewModelu* [29].

5 NÁVRH APLIKACE

5.1 Úrovně API

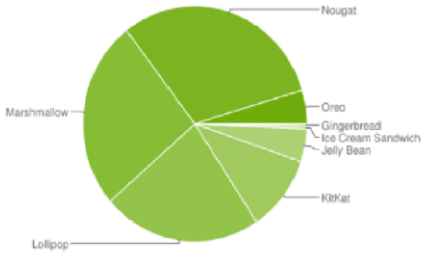
Operační systém Android se neustále vyvíjí s tím, že novější verze mají rozšířenou funkcionalitu, kterou starší verze OS Android nemusí obsahovat. Pří psaní aplikací je potřeba zvolit, které verze OS Android bude daná aplikace podporovat.

5.1.1 Minimum API level

Atribut minimum API level stanovuje nejnižší verzi OS Android, na které může být aplikace spuštěna. Při volbě tohoto atributu je potřeba brát v potaz dva protichůdné faktory, jimiž jsou funkcionalita a množství zařízení, na která může být aplikace nainstalována.

V případě nastavení příliš nízkého parametru Minimum API level bude sice aplikace dostupná většímu počtu zařízení, avšak vývojář nebude mít k dispozici nástroje zavedené v novějších verzích OS Android. Vývojáři tak musí zvolit určitý kompromis a nastavit Minimum API level tak, aby měli k dispozici všechny potřebné funkce a zároveň aby jejich aplikace podporovala co největší počet zařízení. K usnadnění tohoto rozhodování Google pravidelně vydává statistiky využívaných verzí OS Android. V době psaní této práce bylo rozložení verzí OS Android jak je vidět na Obr. 4.

Version	Codename	API	Distribution
2.3.3 - 2.3.7	Gingerbread	10	0.3%
4.0.3 - 4.0.4	Ice Cream Sandwich	15	0.4%
4.1.x	Jelly Bean	16	1.7%
4.2.x		17	2.2%
4.3		18	0.6%
4.4	KitKat	19	10.5%
5.0	Lollipop	21	4.9%
5.1		22	18.0%
6.0	Marshmallow	23	26.0%
7.0	Nougat	24	23.0%
7.1		25	7.8%
8.0	Oreo	26	4.1%
8.1		27	0.5%



Obr. 4: Rozložení verzí OS Android k 27. 4. 2018 [1].

S přihlédnutím na současné rozložení verzí OS Android mezi uživateli byl atribut Minimum API level nastaven na 19 (Android 4.4 Kitkat). Aplikace by tak měla být spustitelná na cca 95 % zařízení, přičemž do budoucna se toto procento bude dále zvyšovat.

5.1.1 Maximum API level

Společnost Google doporučuje vývojářům kromě Minimum API level nastavit také Maximum API level, který určuje, do které verze OS Android je aplikace otestována. Kompatibilita s vyššími verzemi OS Android většinou není problémová, avšak mohou se vyskytnout i výjimky. Problém s kompatibilitou vyšších verzí mohl nastat například u Android 6.0, kdy se změnila pravidla pro schvalování povolení.

Vzhledem k tomu, že aplikace byla testována na zařízení s Android 8.0, byl tento atribut nastaven na 26.

5.2 Nutná povolení

Aplikace mají v systému Android poměrně velkou svobodu, kterou je ale nutné v zájmu ochrany uživatele ošetřit. Pokud chtějí vývojáři aplikací využít nějakou bezpečnostně rizikovou funkcionalitu, musejí uživatele požádat o povolení. Existují dvě skupiny povolení, kritická a nekritická.

Nekritická povolení je nutné schválit pouze při instalaci aplikace. Aplikace vytvářená v rámci této bakalářské práce potřebuje takováto povolení celkem tři:

1) Povolení k přístupu na internet.
2) Povolení ke zjištění stavu síťového připojení.
3) Povolení k zápisu do externích úložišť.

Kritická povolení byla v minulosti schvalována společně s těmi nekritickými při instalaci aplikace, ale od Androidu verze 6.0 musí uživatel tato kritická povolení schválit jednotlivě za běhu aplikace. Uživatel má možnost některá z nich odmítnout a vývojáři s tím musejí počítat. Aplikace psaná v rámci této práce potřebuje dvě kritická povolení:

1) Přístup k přibližnému odhadu polohy.
2) Přístup k přesnému odhadu polohy.

5.3 Volba architektury

Z důvodu narůstající komplexnosti zdrojového kódu bylo nutné implementovat jednu z doporučovaných architektur a rozdělit tak aplikaci na vrstvy. Nakonec byla zvolena architektura MVVM. Důvod této volby je především podpora ze strany společnosti Google, která na konferenci Google I/O 2017 představila sadu nástrojů *Architecture Components*, včetně tříd *ViewModel* a *Livedata*.

Třída *ViewModel* vyřešila problémy s uchováváním stavu logické vrstvy během událostí životního cyklu aktivit a fragmentů (například při změně orientace displeje). Třída *LiveData* pak usnadnila komunikaci mezi vrstvami *ViewModel* a *View* (opět s ohledem na události životního cyklu aktivit a fragment) prostřednictvím vzoru *Observable-Observer* [30].

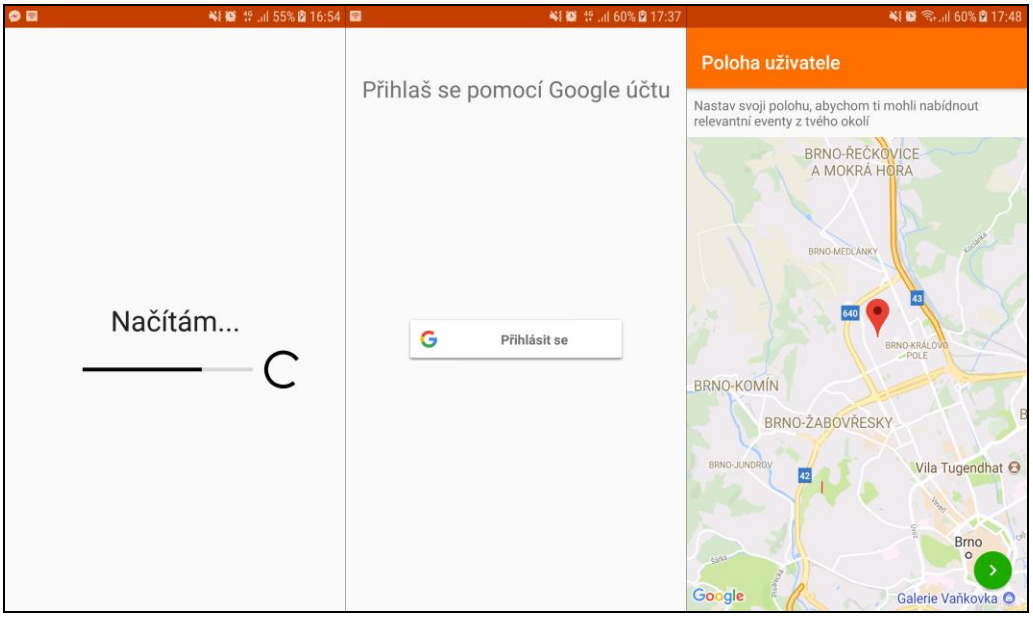
6 REALIZACE

6.1 Hlavní aktivity

V následujících odstavcích je popsána základní funkčnost a základní algoritmy aplikace. V zájmu zachování přehlednosti a pochopitelnosti není brán při popisu algoritmu zřetel na architekturu aplikace. Popis funkcionalit bude rozdělen do bloků svázaných s konkrétní aktivitou.

6.1.1 Launcher

Launcher je první aktivita která se spustí po zapnutí aplikace. Uživatelské rozhraní této aktivity (viz Obr. 5) se skládá se tří prvků – nadpisu „načítám…“ a dvou *ProgressBarů,* přičemž horizontální *ProgressBar* ukazuje aktuální stav inicializace aplikace.



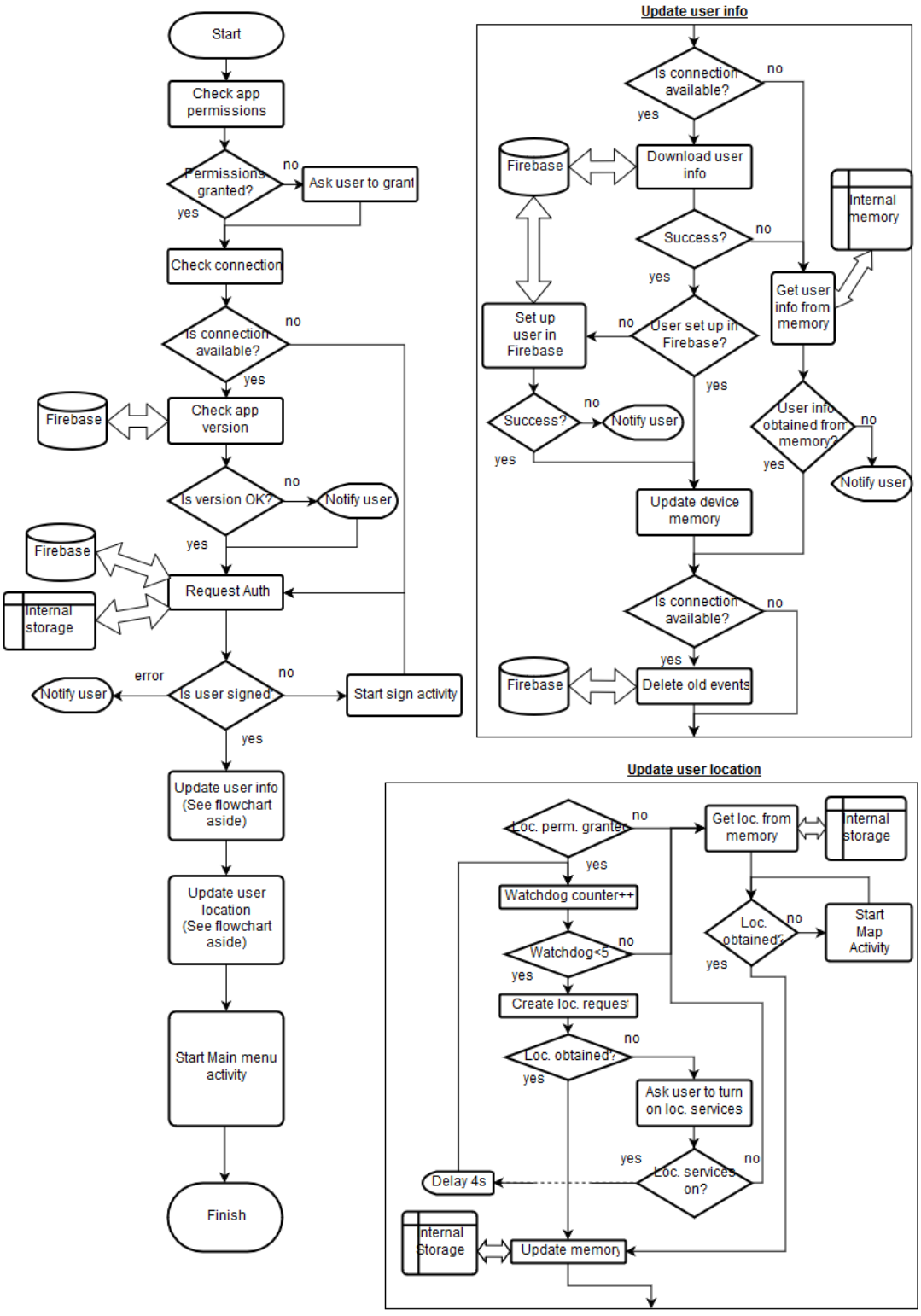
Obr. 5: Uživatelské rozhraní aktivity *Launcher* a přidružených aktivit *Sign* a *Map*.

Hlavní úlohou této aktivity je inicializace aplikace, tedy kontrola a aktualizace všech dat potřebných pro běh aplikace.

Po svém spuštění aktivita nejprve zkontroluje, jestli má uživatel schválená všechna nutná kritická povolení – v tomto případě přístup k poloze uživatele – a v případě, že povolení zatím schválena nebyla, požádá uživatele o schválení. Následně aktivita zkontroluje aktuálnost verze aplikace a v případě zastaralé verze aplikace vyzve uživatele k aktualizaci.

Dalším nezbytným krokem je autentifikace uživatele, respektive jeho přihlášení prostřednictvím Sign aktivity. Jakmile je uživatel přihlášen, aplikace se pokusí

v případě dostupnosti internetu synchronizovat uživatelské informace s databází Firebase. Předposledním krokem je aktualizace uživatelské polohy, která probíhá automaticky, pokud uživatel schválil aplikaci povolení k zjišťování polohy. Pokud tak neučinil, a aplikace nemá v paměti žádnou předchozí uživatelovu polohu, vyzve jej prostřednictvím Map aktivity, aby svoji polohu zadal ručně. Znalost polohy je totiž klíčová k zobrazování relevantních sportovních událostí. Svoji polohu pak uživatel může kdykoliv ručně změnit z aktivity *Events*. Celý průběh inicializace aplikace v aktivitě *Launcher* je vidět na Obr. 6.



Obr. 6: Algoritmus inicializace aplikace.

6.1.2 MainMenu

MainMenu je hlavní rozcestník aplikace. Její uživatelské rozhraní (viz Obr. 7) obsahuje logo, tlačítko pro kontaktování vývojáře a čtyři hlavní tlačítka, prostřednictvím kterých se přistupuje do čtyř hlavních částí aplikace, tedy do sekce sportovních událostí (eventů), skupin, uživatelského profilu a vlastních sportovišť.

Obr. 7: Uživatelské rozhraní aktivity *MainMenu*.

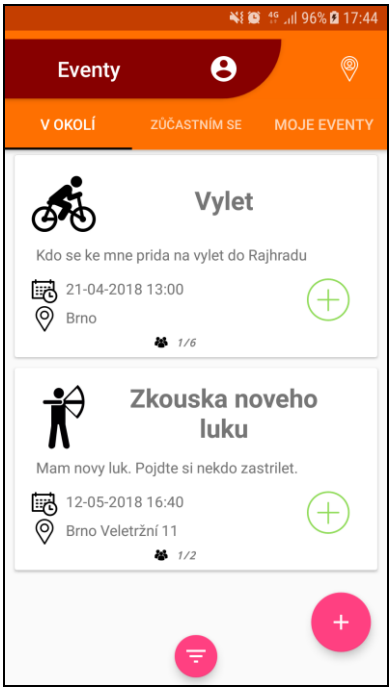
Funkčně se jedná o velmi jednoduchou aktivitu, ke které pro její jednoduchost nebyl přiřazen žádný *ViewModel*. Čtyři hlavní tlačítka spouští příslušné aktivity jednotlivých sekcí, tlačítko „Kontaktuj vývojáře“ zobrazí dialog, prostřednictvím kterého je vyslán *Intent* na spuštění emailového klientu s předem vyplněnou emailovou adresou a předmětem.

6.1.3 Events

Aktivita *Events* je jednou z nejdůležitějších aktivit této aplikace. Její účel je zobrazovat uživateli sportovní události a umožnit mu přihlašovat se na ně, případně se z nich odhlašovat.

Aktivita je typu „*Tabbed aktivity*“ – tedy aktivita se záložkami, mezi kterými se lze pohybovat buď tlačítky pod aplikační lištou, nebo přejetím prstu do strany. Aktivita obsahuje celkem tři záložky, které ukazují sportovní události v okolí, sportovní události na které se uživatel již přihlásil a události které vytvořil. Události jsou zobrazovány pomocí kartiček, na kterých jsou zobrazeny základní informace o události včetně jména, sportu, zkráceného popisu a dalších. Zároveň je na kartičce tlačítko, pomocí kterého se lze k události rovnou přihlásit (v záložce sportovních událostí z okolí), případně se z události odhlásit (v záložce přihlášených událostí). Zobrazení detailu události se provádí přímo kliknutím na konkrétní kartičku. Aktivita zároveň obsahuje plovoucí akční tlačítko, pomocí kterého se spouští aktivita na vytvoření nové události. V záložce

událostí z okolí je pak ještě akční tlačítko, pomocí kterého se filtrují události podle sportu. Nad záložkami se pak nachází ještě *Toolbar* se dvěma tlačítky, pomocí kterých se uživatel dostane do svého profilu, respektive tlačítko na nastavení aktuální uživatelovi polohy. Celkový vzhled aktivity je možné vidět na Obr. 8.



Obr. 8: Uživatelské rozhraní aktivity *Events*.

Po svém spuštění aktivita nejprve zkontroluje připojení k internetu. V případě, že připojení k dispozici není, smaže z vnitřní databáze všechny události z okolí (jedná se o neaktuální sportovní události, které zůstaly v paměti po předchozím spuštění této aktivity). Uživateli jsou ale i nadále k dispozici přihlášené a spravované sportovní události.

V případě, že připojení k internetu je k dispozici, smažou se z vnitřní databáze *Room* všechny sportovní události, aby byla následně vnitřní paměť synchronizována s cloudovou databází Firebase.

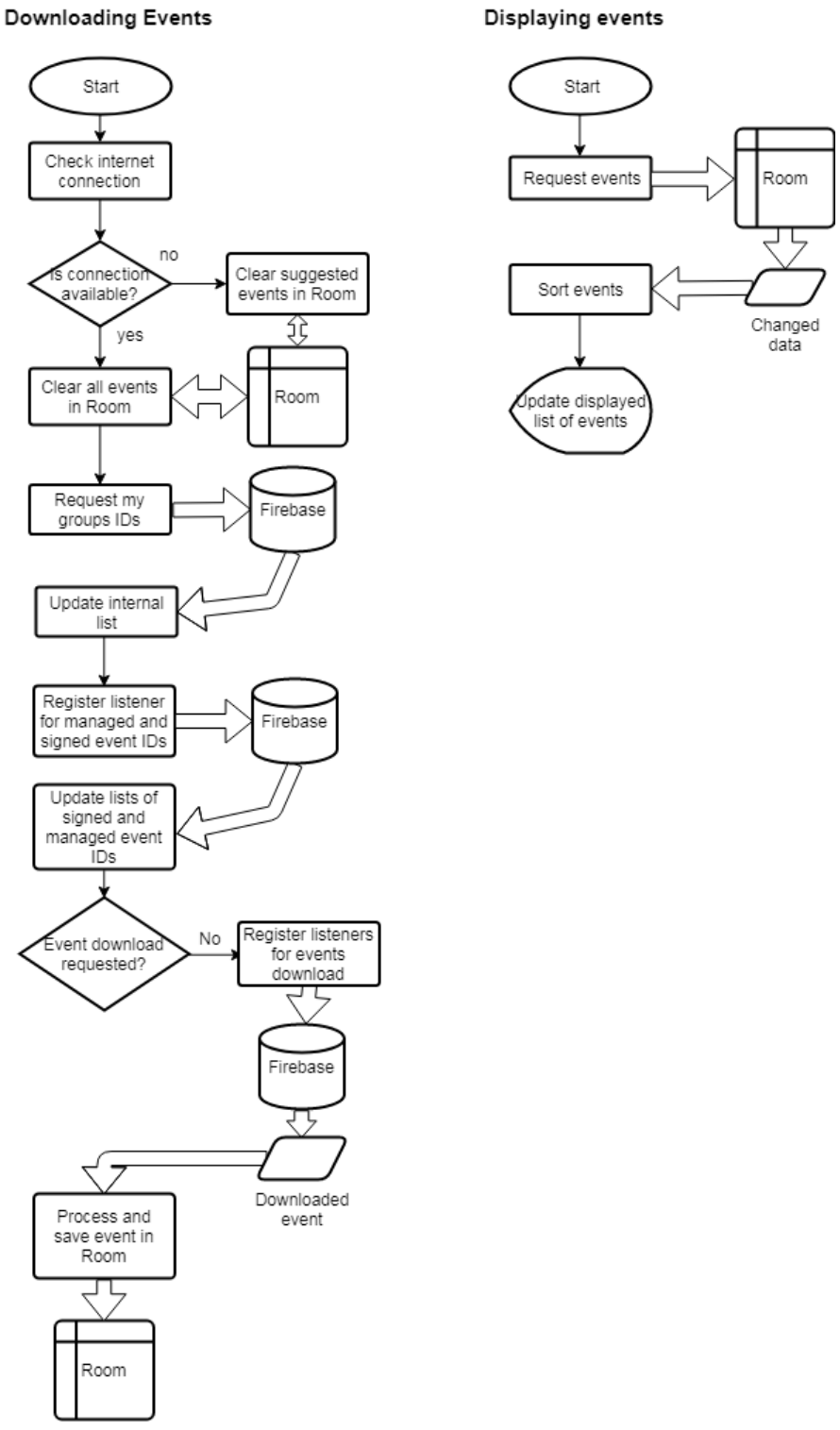
Dalším krokem je stažení seznamu skupin, do kterých je uživatel přihlášen. Tento krok je nutný především z důvodů, aby aplikace určila, které události není uživatel oprávněn vidět (jedná se o události vytvořené v cizích skupinách s viditelností pouze jejím členům).

Následně se registruje *Listener* na ID přihlášených a spravovaných událostí. Kdykoliv se seznam přihlášených a spravovaných událostí na Firebase změní, aktivita je o tom informována a vnitřní seznam si aktualizuje. Pomocí tohoto seznamu aktivita třídí události do příslušných záložek.

Jakmile má aktivita k dispozici seznam ID přihlášených a spravovaných událostí, požádá vrstvu *Model* o registraci *Listenerů* na samotné sportovní události. Poté co vrstva *Model* začne jednotlivé sportovní události vracet, aktivita události roztřídí a uloží do vnitřní databáze *Room*. Na databázi *Room* pak mají registrované *Listenery* jednotlivé záložky ve formě fragmentů. Kdykoliv nastane v databázi *Room* změna, fragmenty

obdrží aktuální seznam sportovních událostí. V záložkách spravovaných a přihlášených událostí jsou události třízeny pouze podle data vzestupně, v záložce událostí z okolí jsou události třízeny nejen podle data, ale i podle vzdálenosti. Jako první se tedy zobrazují události, které se nacházejí blízko uživatele a budou se konat v brzké době.

Celý proces stahování a zobrazování sportovních událostí je naznačen na Obr. 9.



Obr. 9: Průběh aktivity *Events*.

Stahování sportovních událostí

Předtím, než vrstva Model začne stahovat samotné sportovní události, potřebuje rovněž seznam jejich ID.

U přihlášených a spravovaných událostí má na Firebase k dispozici přímo seznam ID, na který registruje *Listener*. Kdykoliv se tedy tento seznam změní, *Model* obdrží informaci o tom, kterého ID se změna týká a zda-li se jedná o přidání nebo smazání události ze seznamu.

U sportovních událostí z okolí je situace složitější, neboť přirozeně neexistuje jejich seznam na Firebase. Požadavky na sportovní události z okolí jsou následující:

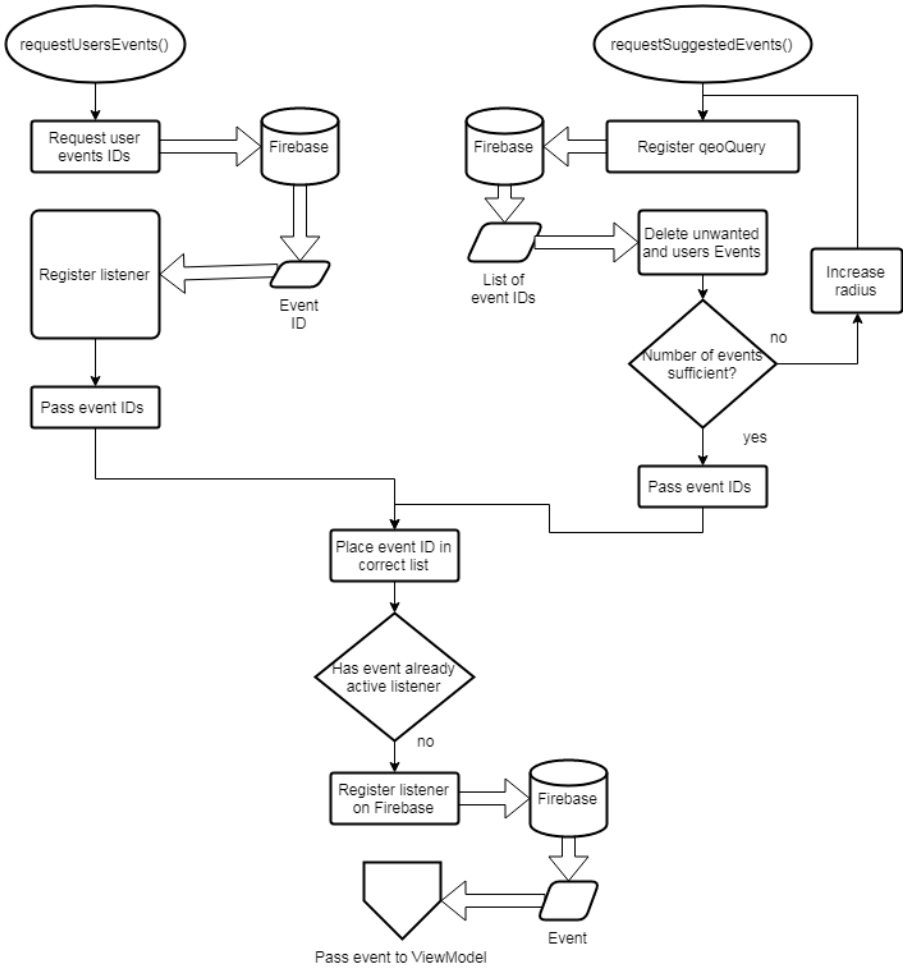
1. Jejich vzdálenost od uživatele by neměla být větší než 50 km.
2. Při vyhledávání je nutné počítat s nastaveným filtrem na sporty.
3. Seznam stažených událostí nesmí být příliš dlouhý.

Prvním problémem, se kterým je potřeba se vypořádat, jsou omezené vyhledávací možnosti Firebase. Sama databáze dokáže vyhledávat pouze podle jediného klíče, což je nedostatečné i na samotné vyhledávání podle polohy. Naštěstí existuje knihovna *Geofire*, která zakóduje zeměpisné souřadnice do jediného klíče, a události tak lze vyhledávat podle polohy stylem zadání středu a poloměru prohledávané oblasti. Knihovna *Geofire* tak vyřešila první požadavek.

Co se týče druhého požadavku, ten je vyřešen vhodnou podobou ID sportovních událostí. ID sportovních událostí se skládají ze sportu, času vytvoření a ID uživatele, který je vytvořil. Ve chvíli kdy knihovna *Geofire* vrátí seznam ID událostí, které se nacházejí v určitém okruhu od uživatele, lze tento seznam profiltrovat a vybrat z něj pouze události, které jsou v souladu s vyhledávaným seznamem sportů, aniž by bylo nutné celé události stahovat. Zároveň jsou ignorovány události uživatele (přihlášené i spravované) a neaktuální události.

Třetí požadavek je řešen variabilní velikostí vyhledávacího okruhu. Vyhledávání začne na okruhu jeden kilometr. V následujícím kroku je zjištěno, kolik chtěných událostí se v tomto okruhu nachází a v případě, že počet není dostatečný (v současné době nastaveno na 200), zvětší se okruh vyhledávání. Takto algoritmus probíhá až do chvíle, kdy okruh vyhledávání dosáhne poloměru 50 km, nebo počet událostí dosáhne stanoveného čísla. Po skončení algoritmu registrovaný *Listener* reaguje na všechny události v daném okruhu, přičemž je notifikován o přidání nebo smazání událostí z tohoto okruhu.

Ve chvíli kdy jsou k dispozici ID chtěných sportovních událostí, jsou tyto identifikační klíče předány další třídě, která řeší stažení celých událostí. Tato třída pak podle přijatých identifikačních klíčů registruje na konkrétní události *Listenery*, a stažené události vrací aktivitě (respektive *ViewModelu* aktivity), která si o ně požádala. Proces stahování sportovních událostí je zobrazen na Obr. 10.



Obr. 10: Algoritmus stahování sportovních událostí.

6.1.4 AddEvent

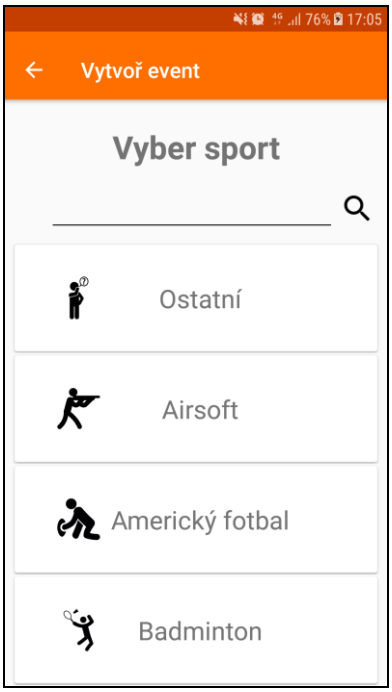
Hlavním posláním této aktivity, jak už název napovídá, je vytvoření nové sportovní události, na kterou by se ostatní mohli hlásit. Do aktivity *AddEvent* je přístup z aktivity *Events* prostřednictvím plovoucího akčního tlačítka (případně z aktivity *GroupDetail*).

Uživatelské rozhraní je velmi jednoduché, prakticky obsahuje pouze *Toolbar* s nadpisem a tlačítkem zpět a kontejner na fragmenty.

Aktivita má jeden společný *ViewModel*, pomocí kterého se nastavují jednotlivé položky události, každý fragment má pak ještě vlastní *ViewModel*, který řeší vnitřní logiku. Postupně se pomocí fragmentu nastavuje sport, nadpis, popis, lokalita, datum, čas a maximální počet účastníků. Na konci procesu tvorby události je pak uživateli k dispozici náhled.

Výběr sportu

Fragment výběr sportu obsahuje *RecyclerView* se seznamem sportů v podobě kartiček, a vyhledávací okno (viz Obr. 11). V současné době je možné vybírat z více než 50 sportů, které jsou seřazeny alfabeticky. Fragment zároveň umožňuje fulltextově vyhledávat sporty ze seznamu prostřednictvím vyhledávacího okna. Názvy i vyhledávání se přizpůsobují aktuální jazykové mutaci.

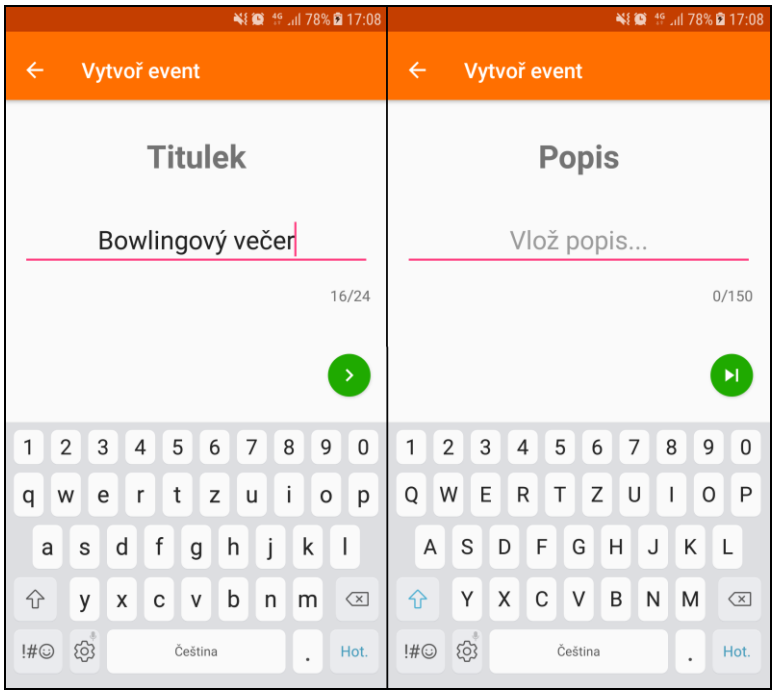


Obr. 11: UI fragmentu Sport.

Nastavení nadpisu a popisu

Nastavení nadpisu a popisu sportovní události probíhá pomocí stejného fragmentu *StringFragment*. Jeho uživatelské rozhraní (viz Obr. 12) obsahuje nadpis, editovatelné textové pole, textové pole na čítání znaků a potvrzující tlačítko.

Nadpis je povinná složka, a tak lze zadat pouze nadpis nenulové délky, popis lze přeskočit. Z důvodů omezeného místa na kartičkách událostí je délka nadpisu omezena na 24 znaků, délka popisu je pak omezená na 150 znaků.



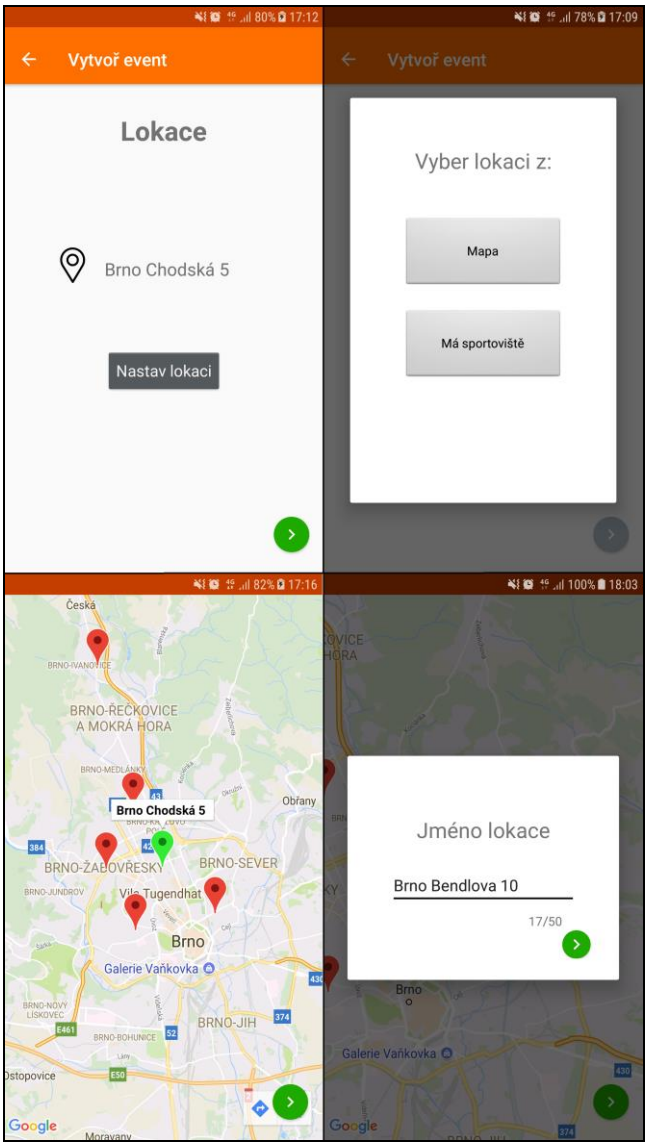
Obr. 12: UI fragmentu *StringFragment*.

Výběr lokace

Fragment obsahuje jednoduché uživatelské rozhraní (viz Obr. 13) s jedním tlačítkem a textovým polem s názvem vybrané lokality.

Jakmile se uživatel dostane na fragment výběru lokace, automaticky se objeví dialogové okno, prostřednictvím kterého si může vybrat, zda lokaci zvolí z mapy nebo ze seznamu oblíbených sportovišť. Mapa se po spuštění automaticky přiblíží na uživatelovu polohu. Zároveň se na mapě objeví uživatelova uložená sportoviště. Lokalitu události uživatel jednoduše volí poklepáním na mapě, případně může zvolit jedno z již uložených sportovišť. Po zvolení místa na mapě a potvrzení zeleným tlačítkem je uživatel vyzván k zadání jména lokality. Pokud je dostatečně kvalitní přístup k internetu, jméno lokality je předvyplněno adresou získanou knihovnou *GeoCoder*. Délka názvu lokality je z důvodu omezeného místa na kartičce události omezené na 50 znaků.

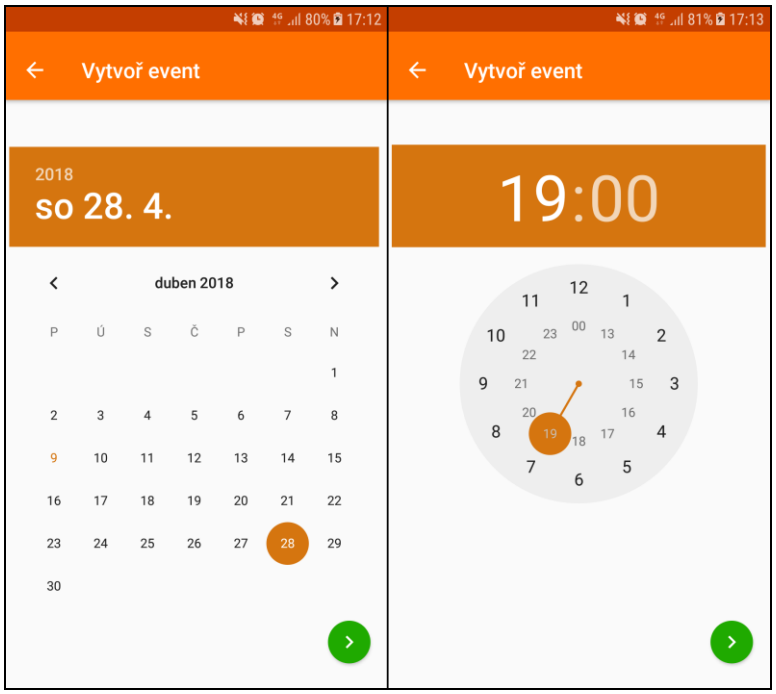
V případě, že uživatel zvolil jedno z již uložených sportovišť, je dialog s nastavením jména lokace přeskočen a uživatel je rovnou navrácen na fragment výběru lokace.



Obr. 13: UI fragmentu *Location* včetně přidružené aktivity Map.

Nastavení data a času

Fragmenty datum a čas používají oficiální widgety pro hodiny a kalendář od společnosti Google (viz Obr. 14). Fragmenty hlídají, aby uživatel zadal čas a datum v budoucnosti, v opačném případě je uživatel upozorněn, aby zadal korektní hodnoty. Čas je ukládán v podobě millis – tedy počet milisekund od 1. 1. 1970 – a interpretován pomocí standardní knihovny *java.Calendar*.



Obr. 14: UI fragmentů Date a Time.

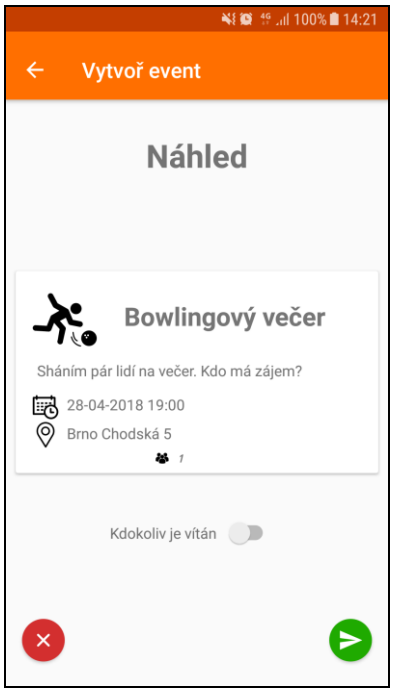
Stanovení maximálního počtu účastníků

U událostí je rovněž možné stanovit maximální počet účastníků. Fragment (viz Obr. 15) obsahuje krom nadpisu dvě tlačítka a okno s ukazatelem. Defaultně je neomezený počet znázorněný znakem nekonečno, který lze změnit bud tlačítky plus a mínus, nebo poklepáním na ukazatel lze zadat číslo na klávesnici.

Obr. 15: UI fragmentu *MaxNumOfUsers*.

Náhled

Posledním fragmentem aktivity *AddEvent* je náhled. Ukazuje událost jako kartičku, která se zobrazuje ostatním v aktivitě *Events*. Pokud je událost přidávána v rámci skupiny, je pod kartičkou ještě možné zvolit, zdali bude událost viditelná pouze členům skupiny, nebo se na ni bude moci přihlásit kdokoliv. Kromě kartičky s událostí a přepínače pak náhled obsahuje dvě další tlačítka, jedním se událost uloží na Firebase, druhým se celá událost zruší. Vzhled fragmentu je vidět na Obr. 16.



Obr. 16: UI fragmentu *OverView*.

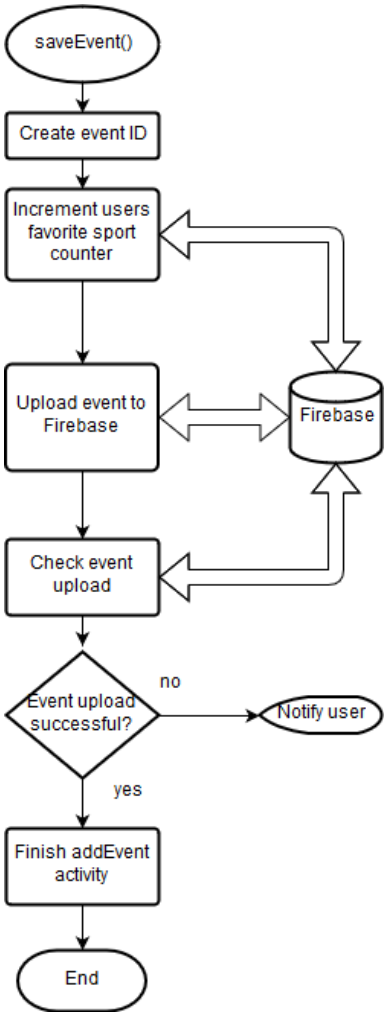
Uložení události na Firebase

Poté co uživatel stiskne zelené tlačítko v *OverView* fragmentu, spustí se funkce ukládání událostí na Firebase. Funkce nejprve vytvoří ID události, které je složeno ze jména sportu, aktuálního času ve formátu millis a ID uživatele, který událost vytvořil. V případě, že se jedná o událost pro skupinu, je do jejího ID přidáno ještě ID dané skupiny.

Jednotlivé položky ID zajistí nejen jednoznačnost ID události, ale především možnost aplikovat filtry při stahování událostí, aniž by se celá událost musela stáhnout. Stačí tedy stáhnout pouze seznam ID událostí, seznam vyfiltrovat a poté už stahovat pouze události, které jsou chtěné.

Dalším krokem je zvýšení čítače uživatelových oblíbených sportů na Firebase, poté už se přistoupí k nahrání samotné události.

Upload události na Firebase probíhá ve dvou krocích. Prvním je samotný upload události, druhým krokem je opětovné stažení a kontrola, že se událost nahrála. Pokud událost nebyla uploadována – uživatel je o tom notifikován pomocí *Toastu* (textu, který se na určitou dobu objeví na obrazovce). Jestliže vše proběhlo správně, automaticky se ukončí aktivita *AddEvent* a uživatel je notifikován opět pomocí *Toastu*, že událost byla úspěšně nahrána na server. Celý proces je vidět na Obr. 17.



Obr. 17: Algoritmus ukládání sportovních událostí na Firebase.

6.1.5 EventDetails

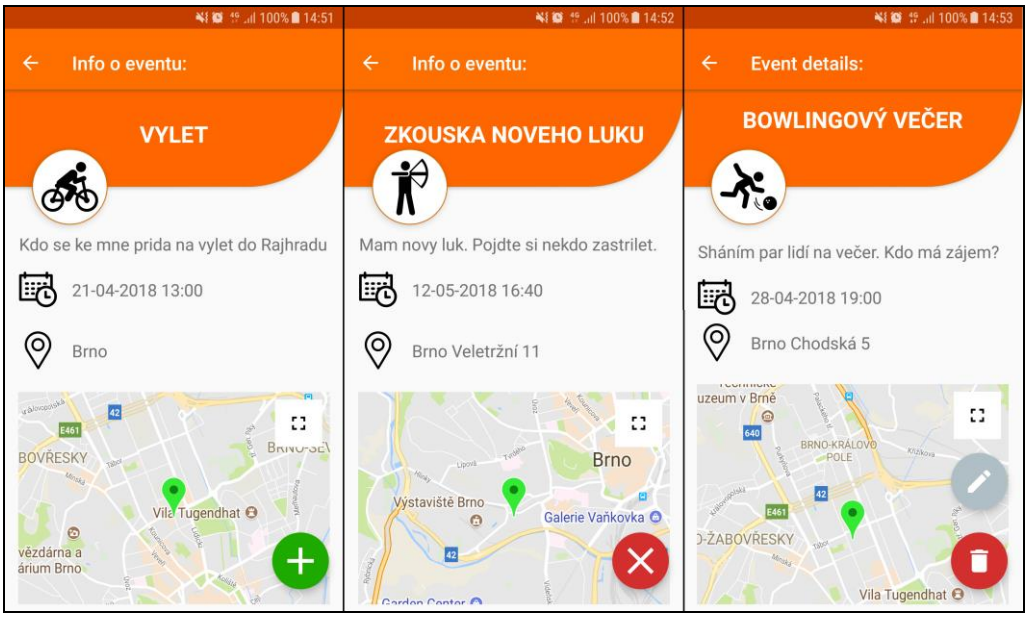
Aktivita *EventDetails* slouží k zobrazení podrobností o jednotlivých událostech. Přistupuje se k ní přímo stisknutím kartičky v aktivitách *Events* nebo *GroupsDetails*.

UI aktivity (viz Obr. 18) obsahuje všechny informace o dané události včetně nadpisu, sportu, plného popisu, místa konání a seznamu přihlášených uživatelů. Místo konání události je zobrazeno nejen jako textové pole s názvem lokace, ale rovněž v mapce.

Aktivita má tři verze – pro přihlášené, pro nepřihlášené a pro spravované události.

Verze pro nepřihlášené události obsahuje plovoucí akční tlačítko, pomocí kterého se lze na události přihlásit, verze pro přihlášené události pak přirozeně obsahuje odhlašovací plovoucí akční tlačítko.

Verze pro spravované události obsahuje plovoucí akční tlačítka dvě, a to pro smazání události a pro úpravu události.



Obr. 18: UI aktivity *EventDetail*.

U verzí pro přihlášené a nepřihlášené události je funkčnost aktivity *EventDetail* velmi jednoduchá. Pomocí akčních tlačítek umožňuje přihlásit se, respektive odhlásit se od události. Dále aktivita umožňuje spustit mapovou aktivitu, na které je daná událost vyznačena. Kromě výše zmíněného pak aktivita může spustit dialog se seznamem přihlášených uživatelů, který se zobrazí prostřednictvím tlačítka nacházejícího se na konci rolovacího okna aktivity.

Verze pro spravované události umožňuje smazání a úpravu události. Upravovat lze veškeré možnosti kromě sportu. Rozhodnutí neumožnit měnit sport bylo provedeno ze dvou důvodů. Za prvé se uživatelé hlásí k událostem především podle sportu, takže ve chvíli, kdy by správce události sport změnil, by se pro ně událost stala irelevantní. Druhý důvod je čistě funkční, neboť sport události je obsažen v ID události, které se nemůže měnit. V případě že bychom změnili sport události, filtrovací algoritmus by událost stále vyhodnocoval podle původního sportu, neboť události filtruje na základě ID.

Úprava události se aktivuje stisknutím horního plovoucího akčního tlačítka, přičemž obě plovoucí tlačítka jsou následně zaměněna za plovoucí tlačítko s ikonou potvrzení.

Následně stačí kliknout přímo na informaci, kterou si uživatel přeje změnit, a objeví se dialog, do kterého lze zadat novou hodnotu položky. Pro nastavení času se objeví sled dialogů data a času, pro nastavení lokace se spustí mapová aktivita s následným dialogem pro zadání jména lokace.

Všechny vstupy jsou ošetřeny stejným způsobem, jako tomu bylo v aktivitě *AddEvent*, tedy textová pole mají minimální a maximální počet znaků a datum musí být zadáno v budoucnosti.

Po stisknutí potvrzovacího plovoucího akčního tlačítka jsou změněné položky na Firebase uloženy podobným způsobem, jako byla událost uploadována v aktivitě *AddEvent*.

6.1.6 Account

Aktivita *Account* (viz Obr. 19) slouží k zobrazení informací o uživatelích. Zobrazované informace zahrnují jméno uživatele, pár slov o uživateli, jeho oblíbené sporty a kontaktní informace. Pokud se jedná o vlastní profil, má navíc uživatel možnost prostřednictvím této aktivity jednotlivá pole upravovat.

K aktivitě se přistupuje buď z hlavního menu, nebo z kteréhokoliv seznamu uživatelů v aplikaci. Uživatelské rozhraní obsahuje textová pole se jménem uživatele, popisem uživatele a s jeho kontaktními informacemi. Dále se v aktivitě nachází 5 polí, která zobrazují pět nejoblíbenějších sportů uživatele. V případě vlastního profilu se zviditelní také plovoucí akční tlačítko pro úpravu profilu.

Obr. 19: UI aktivity *Account*.

Upravování jednotlivých položek probíhá v režimu úprav prostým poklepáním na danou položku. Po poklepání na položku se objeví dialogové okno, kam uživatel zadá novou hodnotu. Kontrolována je minimální a maximální délka, která se u jednotlivých položek liší. U úpravy kontaktních údajů je rovněž kontrolován správný tvar odpovídající telefonnímu číslu, respektive E-mailové adrese.

Všechny změny v profilu uživatel uloží pomocí zeleného plovoucího akčního tlačítka, které v editovacím módu nahradilo původní. Změny jsou uloženy nejen na Firebase, ale rovněž v místním úložišti za pomocí takzvaných *SharedPreferences*.

Položka oblíbených sportů je generována automaticky. Kdykoliv uživatel vytvoří událost nebo se k nějaké události přihlásí, inkrementuje se na serveru u daného uživatele a daného sportu čítač, v případě odhlášení uživatele se čítač dekrementuje.

V profilu je pak automaticky zobrazeno pět nejoblíbenějších sportů, včetně počtů událostí, které uživatel v daném sportovním odvětví absolvoval.

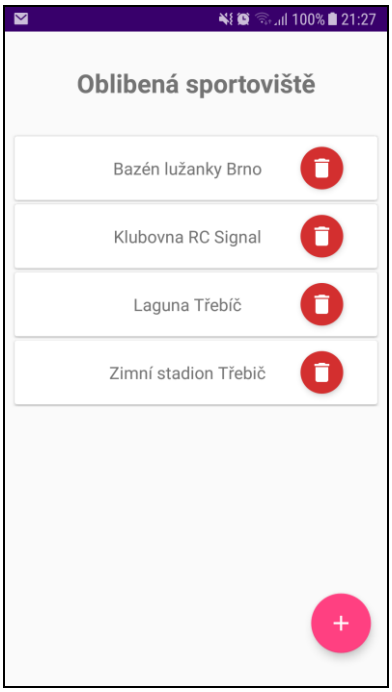
6.1.7 FavoriteSportfields

Aktivita *FavoriteSportfields* slouží k vytváření a výběru oblíbených sportovišť. Původním záměrem bylo vytvořit globální sportoviště viditelná pro všechny uživatele, avšak od toho záměru bylo nakonec upuštěno hned z několika důvodů.

Prvním důvodem je velmi komplikovaná verifikace dat o jednotlivých sportovištích, kde by bylo nutné žádat uživatele o potvrzení pravdivosti údajů. Dalším velkým problémem by pak byla nadbytečná spotřeba dat. Jestliže by se měly zobrazovat globálně viditelná sportoviště na mapě, musely by se všechny informace o nich nejprve stáhnout a až následně třídit. Firebase neumí vyhledávat data podle víc než jednoho klíče (s výjimkou polohy pomocí *GeoFire*), třízení by tak mohlo probíhat až přímo v zařízení. U velkých měst s potenciálně několika tisíci sportovišti by to mohl být značný problém nejen pro uživatele, ale i pro provozovatele aplikace, neboť s většími datovými toky rostou zároveň náklady za provoz databáze Firebase.

Kvůli výše zmíněným problémům bylo nakonec rozhodnuto, že seznam oblíbených sportovišť si každý uživatel bude tvořit sám. Nejen, že se tím eliminují problémy s globálními sportovišti, ale zároveň bude pro uživatele mapa přehlednější, neboť se na ni budou zobrazovat pouze sportoviště, která hodlá využívat. Jediným negativem tohoto řešení pak zůstává to, že uživatel se prostřednictvím aplikace nedozví o sportovištích v okolí.

Samotná aktivita (viz Obr. 20) se skládá ze seznamu sportovišť v podobě *RecyclerView* a plovoucího akčního tlačítka na vytvoření nového oblíbeného sportoviště. Sportoviště jsou zobrazována v podobě karet s názvem sportoviště a s tlačítkem na jeho smazání.



Obr. 20: UI aktivity *FavoriteSportfields*.

Při spuštění aktivity se oblíbená sportoviště nejprve stáhnou a následně zobrazí v *RecyclerView*. Vzhledem k tomu, že bez přístupu k internetu nemá uživatel jak sportoviště využít, nejsou stažená sportoviště ukládána do vnitřní paměti, jak je tomu u událostí nebo skupin.

Přidávání nových sportovišť probíhá obdobným způsobem, jakým probíhá například výběr lokace prostřednictvím mapy při vytváření událostí. Na mapě uživatel vybere lokaci sportoviště a v následujícím kroku zadá prostřednictvím dialogu název sportoviště. V dialogu se zobrazí návrh názvu v podobě adresy sportoviště, avšak pouze za předpokladu, že uživatel má dostatečně rychlý internet a stihly se kompletně načíst mapové podklady. Maximální délka názvu je 50 znaků, minimální délka je pak 1 znak.

Po potvrzení se sportoviště uloží na Firebase a následně se zpětně stáhne a zobrazí v zařízení.

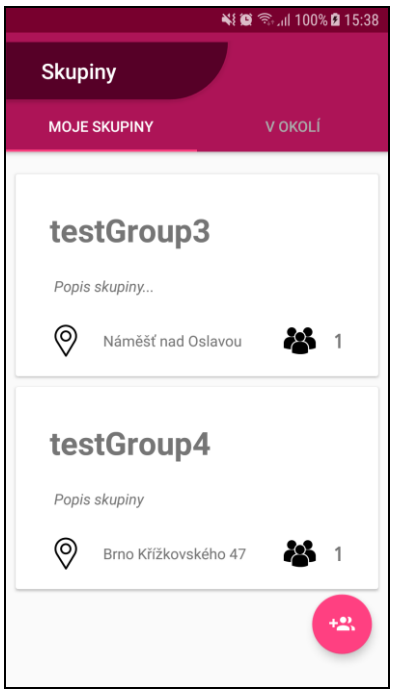
Uložená sportoviště jsou při přidávání nových sportovních událostí přístupná nejen v podobě seznamu, ale jsou rovněž zobrazována na mapě.

6.1.8 Groups

Sportovci se často člení do různých skupin, organizací a kroužků a z tohoto důvodu je v aplikaci dostupná sekce pro skupiny. Aktivita *Groups* je vstupní bránou do této sekce. Přistupuje se k ní z hlavního menu stiskem příslušného tlačítka. Hlavním cílem aktivity je poskytovat uživateli seznam jeho skupin a navrhovat mu skupiny z okolí.

Aktivita je typu „*Tabbed aktivity*“ podobně jako *Events*, tentokrát ale pouze se dvěma záložkami, „moje skupiny“ a „v okolí“. Skupiny jsou zobrazovány opět formou kartiček se základními informacemi. V aktivitě je implementováno plovoucí akční tlačítko,

pomocí kterého se spouští aktivita na vytvoření skupiny. Detaily skupin se zobrazí kliknutím přímo na kartičku. Vzhled aktivity lze vidět na Obr. 21.



Obr. 21: UI aktivity *Groups*.

Chod aktivity je velmi podobný aktivitě *Event*. Po spuštění nejprve zkontroluje připojení k internetu a v případě, že internet není k dispozici, z vnitřní databáze *Room* smaže všechny skupiny, kterých uživatel není členem. Pokud je internet k dispozici, smažou se všechny skupiny a vydá se požadavek na Firebase o stažení uživatelových skupin a skupin v blízkém okolí. Firebase vrací skupiny jednotlivě, přičemž u každé skupiny se kontroluje, zdali má všechny potřebné informace a v kladném případě se uloží do databáze *Room*. Zobrazování skupin je řešeno pomocí registrovaného *Observeru* na databázi *Room*, veškeré změny v databázi *Room* se tak automaticky projeví v UI.

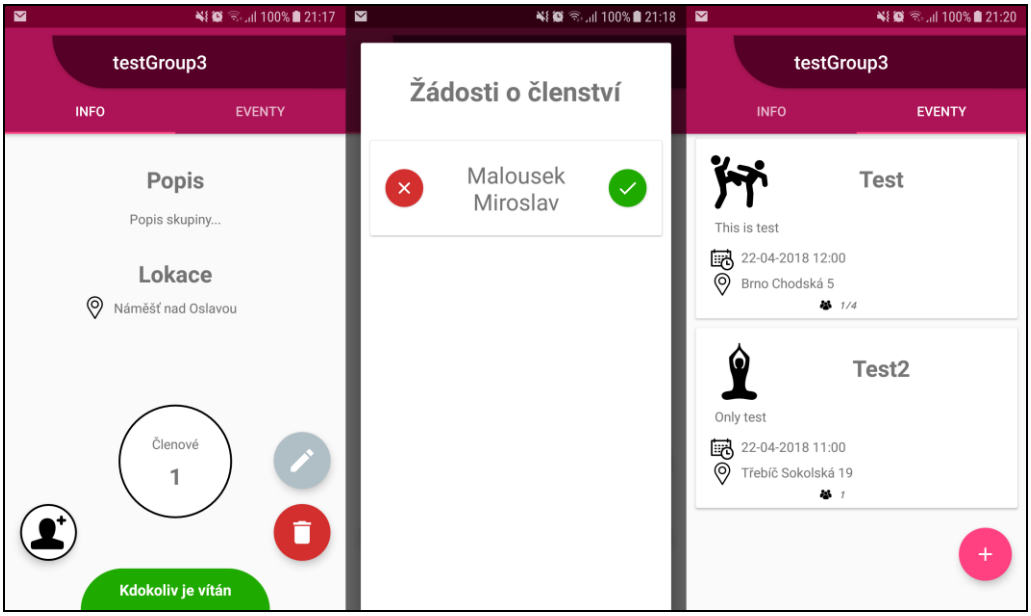
6.1.9 AddGroup

Skupiny je samozřejmě také nutné vytvářet a k tomu slouží aktivita *AddGroup*. Protože Skupiny a sportovní události sdílejí stejné parametry (název, popis, lokace), jsou i aktivity pro jejich vytváření velmi podobné.

Při vytváření skupin lze nastavit jméno, popis a lokaci skupiny, přičemž v tomto pořadí za sebou následují i fragmenty, pomocí nichž jsou tato pole nastavována. V posledním fragmentu, ve kterém je náhled události, pak lze nastavit podmínky pro přijetí do skupiny („kdokoliv je vítán“ a „pouze na žádost“).

6.1.10 GroupsDetail

Aktivita *GroupDetail* (viz Obr. 22) zobrazuje nejen informace o konkrétní skupině, ale slouží i ke správě sportovních událostí dané skupiny.



Obr. 22: UI aktivity Group Detail.

Aktivita je typu „*Tabbed aktivity*“ se dvěma záložkami. První záložka slouží k zobrazení podrobností o skupině, druhá záložka slouží k zobrazení sportovních událostí skupiny. V záložce informace o skupině je zobrazen popis, lokace skupiny, počet členů a podmínky přijetí do skupiny. Po kliknutí na tlačítko zobrazující počet členů se zobrazí seznam, prostřednictvím kterého se lze dostat na profily členů skupiny.

Záložka dále obsahuje různý počet plovoucích akčních tlačítek s různou funkcionalitou závislou na vztahu skupiny a daného uživatele. Pro uživatele nepřihlášené k dané skupině je k dispozici tlačítko na odeslání žádosti o přijetí, případně přímo na vstoupení do skupiny.

Pro normální členy skupiny jsou k dispozici dvě plovoucí akční tlačítka, jedním se ze skupiny odchází, druhé tlačítko zobrazí seznam žádostí o vstup do skupiny, přičemž jakýkoliv člen skupiny má právo potvrdit žádost o členství jinému uživateli.

Správce skupiny (uživatel, který skupinu vytvořil) má pak k dispozici plovoucí akční tlačítka tři. První stejně jako u členů slouží ke schvalování žádostí o vstup, zbylá dvě slouží k úpravě a ke smazání skupiny.

V záložce „Eventy“ jsou zobrazovány sportovní události skupiny. Události jsou opět zobrazovány v podobě kartiček s údaji včetně tlačítek na přihlašování a odhlašování. Události se ale zobrazují pouze členům skupiny. Z této záložky pak lze vytvářen nové sportovní události pomocí plovoucího akčního tlačítka. Přidávání událostí probíhá pomocí aktivity *AddEvent*, pouze v posledním fragmentu sloužícímu

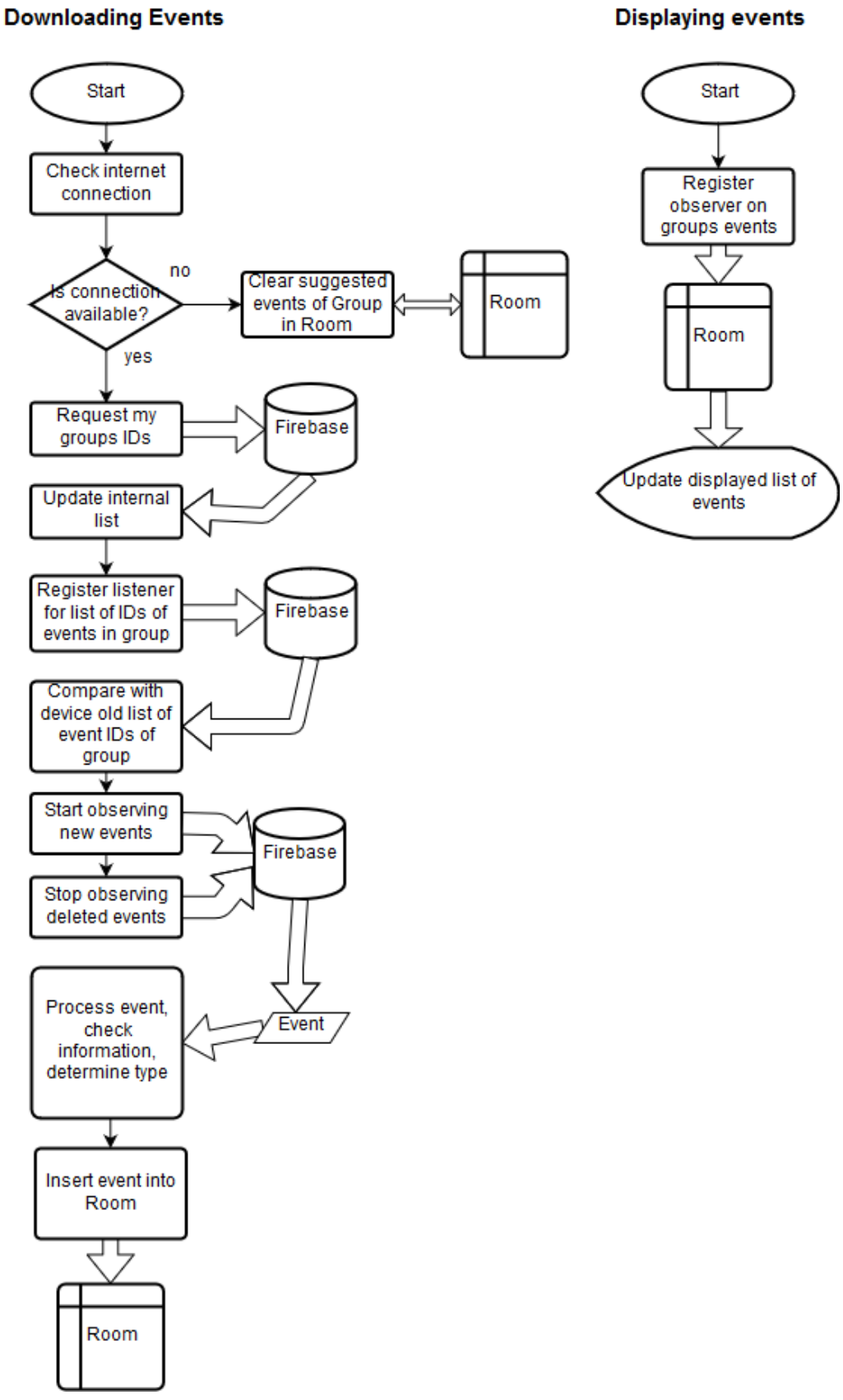
k náhledu události se objeví možnost nastavení viditelnosti události (pouze pro skupinu, pro všechny). Sportovní událost, vytvořená v rámci skupiny, se bude objevovat jak v této záložce, tak i v aktivitě *Events*, přičemž pokud není akce veřejná, zobrazovat se bude pouze členům skupiny.

Co se tyká vnitřního fungování aktivity, tak po spuštění má aktivita všechny informace o skupině k dispozici, neboť ty se stáhnuly již v aktivitě *Groups*. Záložka Info tak pouze inicializuje jednotlivé komponenty UI a jednotlivá pole zaplní příslušnými informacemi.

Události jsou v rámci skupiny stahovány podobným způsobem jako v aktivitě *Events*. Nejdříve se zkontroluje připojení k internetu. Pokud internet k dispozici není, smažou se z interní databáze *Room* všechny události, na které není uživatel v rámci skupiny přihlášen nebo které nevytvořil. V případě dostupného internetu se registruje *Listener* na Firebase, který vrací seznam spravovaných a přihlášených událostí uživatele. Tento seznam je později použit k determinování typu události (přihlášená, spravovaná, nebo navrhovaná). Následně se na Firebase registruje *Listener* vracející seznam událostí skupiny. Seznam se porovná se seznam uvnitř zařízení, přičemž na nové sportovní události je registrován *Listener* na Firebase, u událostí které byly na Firebase smazány se odregistruje *Listener* a následně se událost smaže z vnitřní databáze *Room*.

Jednotlivé události jsou již stahovány asynchronně, přičemž se stáhnou poprvé, kdy se k nim zaregistruje *Listener* a následně pokaždé, kdykoliv jsou na Firebase upraveny. Po stažení události proběhne nejprve kontrola, jestli událost obsahuje potřebná data, a následně se pomocí dříve staženého seznamu spravovaných a přihlášených ID událostí určí její typ. Poté je již událost vložena do vnitřní databáze *Room*.

Zobrazování událostí probíhá naprosto obdobným způsobem jako v aktivitě *Events*. Na databázi *Room* je zaregistrován *Observer* na události dané skupiny, který asynchronně vrací události, kdykoliv nastane změna. Události jsou následně zobrazeny pomocí *RecyclerView*. Celý proces je zobrazen na Obr. 23.



Obr. 23: Algoritmus stahování a zobrazování událostí v aktivitě *GroupDetail*.

6.2 Implementace MVVM

Jak již bylo řečeno v kapitole Návrh aplikace, pro zlepšení přehlednosti a stabilnosti kódu byla použita architektura MVVM, čímž došlo k jasnému oddělení mezi UI vrstvou, logickou vrstvou a datovou vrstvou.

UI vrstvu představují v aplikaci aktivity a fragmenty, kdy ke každé aktivitě, respektive fragmentu (vyjma triviálních), je přidružen jeden *ViewModel*. Úkolem aktivit je nejdříve inicializovat veškeré UI komponenty a následně požádat *ViewModel* o data, ve většině případů asynchronně pomocí objektů *LiveData*. Poté co *ViewModel* potřebná data obstará, aktivita si je odebere a zobrazí. Veškerá logika je tak v aktivitách omezena na minimum. Kromě vizualizace dat provádějí aktivity rovněž spouštění dalších aktivit, dialogových oken a další operace vyžadující objekt typu *Context*, který je k dispozici pouze aktivitám, přičemž jeho předávání *ViewModelům* je považováno za nevhodné.

Logickou vrstvu aplikace představují *ViewModely*. Ke každé složitější aktivitě je přidružen jeden *ViewModel*, k aktivitám obsahujícím více fragmentů (*AddEvent*, *AddGroup*) jsou přidruženy další *ViewModely* pokrývající funkčnost jednotlivých fragmentů. UI vrstva komunikuje z *ViewModelem* prostřednictvím veřejných metod, přímá komunikace v opačném směru probíhat nemůže, protože *ViewModel* nemá na UI vrstvu žádnou referenci. *ViewModel* tak pouze připravuje data pro UI vrstvu do poměných typu *LiveData*, které UI vrstva odebírá.

Mezi hlavní úkoly *ViewModelů* patří žádání o data, jejich zpracování a příprava pro vrstvu UI. Velkou výhodou *ViewModelů* je, že mají podstatně jednodušší životní cyklus než aktivita, přestože jsou s aktivitou svázány. Zatímco aktivita se například při změně orientace restartuje, *ViewModel* zůstává stále ve stejném stavu až do chvíle, kdy je přidružená aktivita ukončena, díky čemuž nedojde při změně orientace ke ztrátě dat, která by bylo nutné jinak zálohovat.

Přístup k datům, a to ať už se jedná o data na *Firebase* či ve vnitřní databázi *Room*, je zajištěn prostřednictvím *Modelů*. Aplikace obsahuje několik *Modelů* rozdělených podle jejich funkce se snahou o dodržení Single-Responsibility principu. *ViewModely* a *Modely* spolu komunikují prostřednictvím interfaců, které musí obě vrstvy implementovat. Díky tomu, že *Modely* jsou navázány na *ViewModely* a nikoliv na UI vrstvu, nedojde při restartování aktivity například při zmíněné změně orientace displeje, k jejich zničení *Garbage Collectorem*.

Aby *Modely* neblokovaly hlavní vlákno aplikace při internetové komunikaci nebo při komunikaci s vnitřní databází *Room*, jsou operace v nich vykonávané přesunuty do jiných vláken pomocí knihovny *RxJava2*, která využívá návrhový vzor *Observable-Observer*.

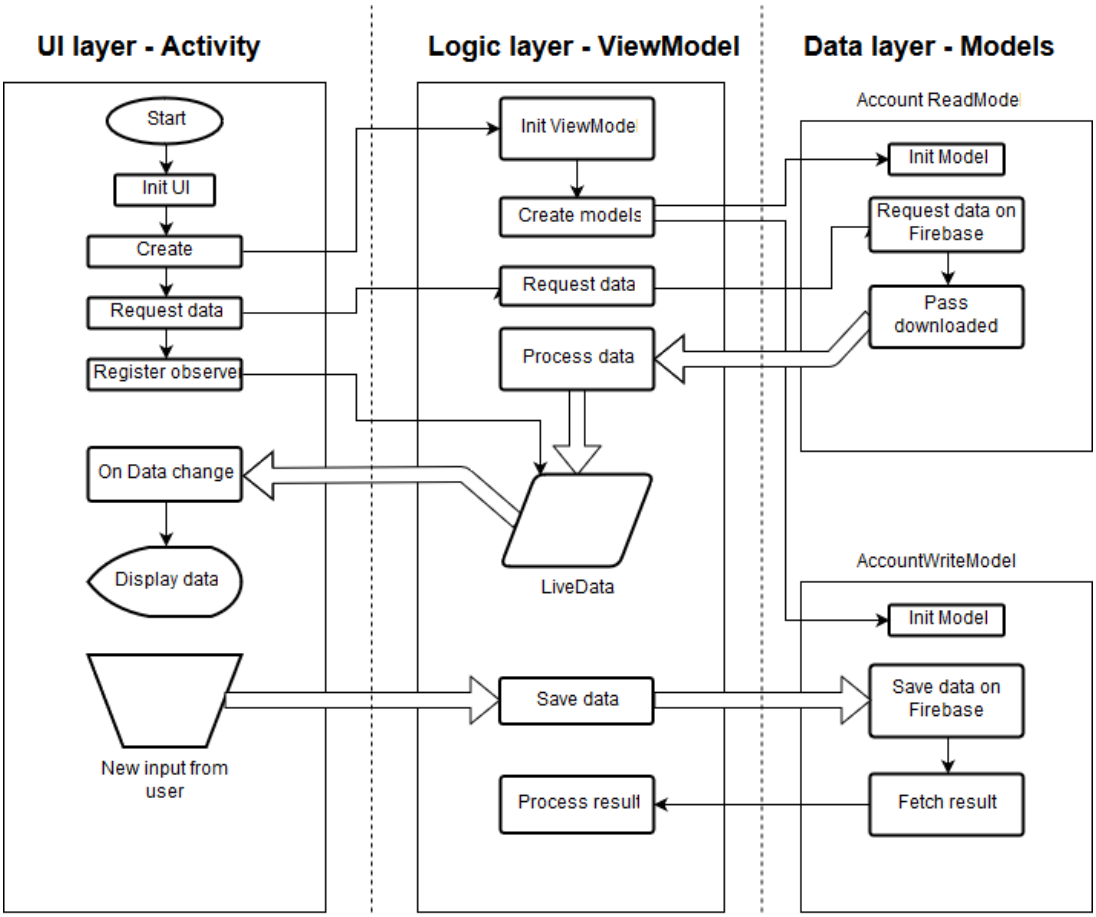
Příklad komunikace mezi vrstvami

Představme si situaci, kdy si uživatel přeje změnit informace ve svém profilu.

Nejdříve se spustí aktivita Account, která inicializuje jednotlivá pole a vytvoří vlastní *ViewModel*. *ViewModel* po své inicializaci vytvoří *Modely* *FirebaseUserReadModel* a *FirebaseUserWriteModel*. Následně aktivita požádá *ViewModel* o data, a registruje Observer na *LiveData* objekt ve *ViewModelu*. *ViewModel* požádá o uživatelská data *Model FirebaseUserReadModel*, který je po obdržení pošle zpět *ViewModelu*. Ten je následně zpracuje a uloží do objektu *LiveData*.

Jakmile nastane v objektu *LiveData* změna, automaticky jsou notifikování všichni, kdo jsou přihlášeni k odběru prostřednictvím zaregistrovaného *Observeru*, v tomto případě aktivita *Account*. Ta tímto způsobem obdrží data, která zobrazí.

Uživatel následně začne upravovat svůj profil. Veškeré změny jsou zachyceny aktivitou a ihned odeslány *ViewModelu*, který změny ukládá do objektu User. Jakmile uživatel stiskne tlačítko uložit, aktivita spustí ukládací funkci ve *ViewModelu*. *ViewModel* následně požádá *Model FirebaseUserWrite* o aktualizaci uživatelského profilu na *Firebase* a odešle mu objekt typu User. *Model* následně informuje *ViewModel* o tom, jestli bylo nahrání úspěšné. Celá operace je zobrazena na Obr. 24.



Obr. 24: Příklad komunikace mezi vrstvami v rámci architektury MVVM.

Díky tomuto přístupu je relativně snadné dělat změny v programu. Například v případě přechodu na jinou cloudovou platformu než Firebase stačí pouze napsat nové *Modely* řešící stahování dat, zbytek aplikace může zůstat beze změn.

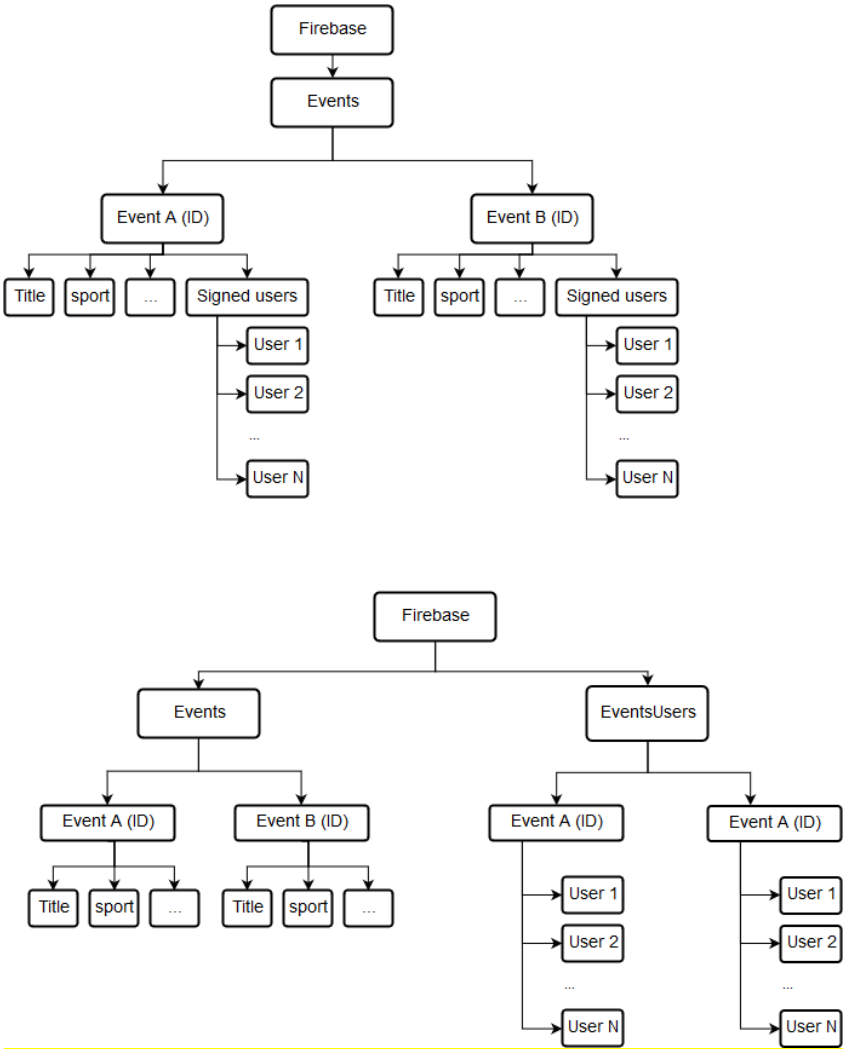
6.3 Implementace Firebase

Jako každý systém má i *Firebase* svá omezení. Prvním již zmíněným omezením je možnost vyhledávání pouze podle jednoho klíče, což při složitějších vyhledáváních zvyšuje nároky na koncová zařízení, neboť výsledná filtrace musí probíhat až na nich.

Data na *Firebase* jsou uložena ve stromové struktuře, s čímž souvisí další velké omezení. Data se stahují formou *listenerů* k určitému „bodu“ této stromové struktury, přičemž stáhne se vždy vše, co k tomuto bodu náleží.

V případě, že by byla vytvořena větev sportovní události se všemi informacemi jako název, popis, sport, čas a další, a současně pod touto větví by byl uložen i seznam přihlášených uživatelů, není žádná možnost jak stáhnout pouze data o události bez seznamu uživatelů. V případě, že by se jednalo o velkou událost, jako například maratón s několika tisíci účastníky, mohlo by to vést k nadměrné spotřebě paměti a dat.

Z tohoto důvodu je nutné implementovat co nejplošší datovou strukturu. To znamená místo jedné větve pro sportovní událost vytvořit větví několik, přičemž v první větvi budou například informace o události, v druhé větvi přihlášení uživatelé, v další větvi lokalizační data pro knihovnu *GeoFire* atd. Díky tomuto přístupu jsou vždy stahována pouze data, která aktuálně aplikace potřebuje. Zjednodušená ukázka hluboké a ploché struktury dat je ukázána na Obr. 25.



Obr. 25: Přiklad hluboké (horní část) a ploché (spodní část) struktury dat na Firebase.

6.4 Metriky kódu

O komplexnosti aplikace do určité míry vypovídá metrika (rozsah) kódu. Aplikace se skládá převážně ze dvou typů souborů: *java* a *xml*. Soubory typu *java* určují chování programu, jedná se tedy o logickou část aplikace. Aplikace se skládá z celkem 129 souborů *java*, které mají celkem 15 342 řádků.

Grafická stránka aplikace je popsána v souborech typu *xml*. Těchto souborů aplikace obsahuje celkem 176 s celkovou délkou 9177 řádků. Velká část kódu v *xml* souborech je ale generována automaticky.

7 ZÁVĚR

V rámci bakalářské práce byla vytvořena funkční aplikace pro operační systém Android, která slouží k domlouvání se na společné sportovní aktivity. Při psaní aplikace bylo dbáno na využívání ověřených programátorských praktik jako je implementace architektury MVVM nebo využívání prvků reaktivního programování za pomocí knihovny *RxJava2*.

Vytvořená aplikace je plně funkční a relativně stabilní. Fungovat by měla na všech zařízeních s operačním systémem Android 4.4 a vyšším. Přihlášení do aplikace probíhá přes Google účet, další možnosti jako Facebook, Twitter a jiné budou přidány v budoucnu.

Splněny byly všechny body zadání s výjimkou přidávání nových druhů sportů uživateli. V průběhu vývoje aplikace bylo rozhodnuto, že uživatel nebude mít právo přidávat nové typy sportů, a to především z toho důvodu, že by pro nový sport chyběla ikonka. V případě, že uživatelům bude chybět nějaký sport či funkcionalita, mají přímo v aplikaci možnost kontaktovat vývojáře prostřednictvím E-mailu.

Nad rámec zadání byla v aplikaci přidána možnost vytvářet a spravovat skupiny, díky čemuž je aplikace vhodná pro různé sportovní kluby, kroužky a další organizace. Z důvodů lepší využitelnosti aplikace obsahuje i základní notifikace, kdy je uživatel upozorněn na jakoukoliv přihlášenou nebo vytvořenou událost hodinu před tím, než se bude událost konat.

Aplikace je v době psaní této práce dostupná v obchodu Google Play v otevřeném beta testovacím režimu. Aplikaci si tak může stáhnout kdokoliv, kdo zadá do vyhledavače její název – Meet2sport.

Aplikace je dostupná v celkem čtyřech jazycích: v angličtině, češtině, slovenštině a němčině, přičemž jazyk se volí automatiky podle nastavení telefonu. Defaultním jazykem aplikace je angličtina.

Do budoucna bude potřeba aplikaci optimalizovat pro různé velikosti a rozlišení displeje. Současná podoba uživatelského rozhraní je optimalizována na mobilní telefon běžných rozměrů (kolem 5 palců) a rozlišení full HD. U jiných úhlopříček může docházet například k částečnému překrytí některých grafických komponent, případně k neestetickým roztažením některých grafických prvků.

Dále je v plánu do aplikace přidat možnost Chatu, která zatím nebyla z časových důvodů do aplikace přidána.

Velká pozornost pro budoucí vývoj bude patřit pravděpodobně cloudovým funkcím Firebase, které byly v době psaní této bakalářské práce teprve v beta-verzi. Cloudové funkce v budoucnu umožní provádění některých automatických operací jako je například mazaní starých událostí na straně serveru.

LITERATURA

[1] *Android Developer* [online]. [cit. 2017-10-03]. Dostupné z: https://developer.android.com/

[2] DRAKE, Joshua J. *Android hacker's handbook*. Indianapolis, IN: John Wiley, 2014. ISBN 978-1-118-60864-7.

[3] BURD, Barry. *Android application development all-in-one for dummies*. 2nd edition. Hoboken, NJ: For Dummies, a Wiley Brand, 2015. --For dummies. ISBN 1118973801.

[4] Android - Application Components. *Tutorials point* [online]. [cit. 2017-10-07]. Dostupné z: https://www.tutorialspoint.com/android/android_application_components.htm

[5] Vyvíjíme pro Android: Intenty, intent filtry a permissions. *Zdrojak* [online]. [cit. 2017-11-01]. Dostupné z: https://www.zdrojak.cz/clanky/vyvijime-pro-android-intenty-intent-filtry-a-permissions/

[6] Android programování - Životní cyklus a nový projekt. *IT NETWORKS* [online]. [cit. 2017-10-08]. Dostupné z: https://www.itnetwork.cz/java/android/tutorial-programovani-pro-android-v-jave-zivotni-cyklus-a-novy-projekt

[7] HELLMAN, Erik. *Android programming: pushing the limits*. Chichester, West Sussex: Wiley, 2014. ISBN 9781118717370.

[8] Android - Architecture. *Tutorials point* [online]. [cit. 2017-11-03]. Dostupné z: https://www.tutorialspoint.com/android/android_architecture.htm

[9] Google is Mandating Linux Kernel Versions in Android Oreo. *XDA Developers* [online]. [cit. 2017-11-03]. Dostupné z: https://www.xda-developers.com/google-mandating-linux-kernel-versions-android-oreo/

[10] Úvod do jazyka Java. *IT NETWORKS* [online]. [cit. 2017-11-03]. Dostupné z: https://www.itnetwork.cz/java/zaklady/java-tutorial-uvod-do-jazyka-java

[11] An Overview of the Android Architecture. *Techotopia* [online]. [cit. 2017-11-03]. Dostupné z: http://www.techotopia.com/index.php/An_Overview_of_the_Android_Architecture

[12] Inside the different Android Versions. *Android Central* [online]. [cit. 2017-11-03]. Dostupné z: https://www.androidcentral.com/android-versions

[13] History. *Android* [online]. [cit. 2017-11-03]. Dostupné z: https://www.android.com/history/

[14] The history of Android OS: its name, origin and more. *Android Authority* [online]. [cit. 2017-11-03]. Dostupné z: https://www.androidauthority.com/history-android-os-name-789433/

[15] Firebase: krátké seznámení. *Zdroják* [online]. [cit. 2017-11-03]. Dostupné z: https://www.zdrojak.cz/clanky/firebase-kratke-seznameni/

[16] Firebase [online]. [cit. 2017-11-03]. Dostupné z: https://firebase.google.com/docs/

[17] Material Design [online]. [cit. 2017-11-23]. Dostupné z: https://material.io/guidelines/#

[18] 7 Steps To Room [online]. [cit. 2018-04-06]. Dostupné z: https://medium.com/google-developers/7-steps-to-room-27a5fe5f99b2

[19] Building database with Room Persistence Library [online]. [cit. 2018-02-05]. Dostupné z: https://medium.com/@ajaysaini.official/building-database-with-room-persistence-library-ecf7d0b8f3e9

[20] Android Architecture Components: Room — Introduction [online]. [cit. 2018-02-24]. Dostupné z: Android Architecture Components: Room — Introduction

[21] What is Reactive Programming? [online]. [cit. 2018-04-06]. Dostupné z: https://medium.com/@kevalpatel2106/what-is-reactive-programming-da37c1611382

[22] Intro [online]. [cit. 2018-04-06]. Dostupné z: http://reactivex.io/intro.html

[23] RxJava 2 – Flowable [online]. [cit. 2018-04-06]. Dostupné z: http://www.baeldung.com/rxjava-2-flowable

[24] Dependency Injection: motivace. Zdroják [online]. [cit. 2018-04-06]. Dostupné z: https://www.zdrojak.cz/clanky/dependency-injection-motivace/

[25] Friendly Introduction to Dagger 2 [online]. [cit. 2018-04-06]. Dostupné z: https://www.zdrojak.cz/clanky/dependency-injection-motivace/

[26] Dagger 2 for Android Beginners — Introduction [online]. [cit. 2018-04-06]. Dostupné z: https://medium.com/@harivigneshjayapalan/dagger-2-for-android-beginners-introduction-be6580cb3edb

[27] Model-View-Presenter: Android guidelines [online]. [cit. 2018-04-06]. Dostupné z: https://www.citacepro.com/dok/BcQAgs2HWC2emHf6

[28] MVP for Android: how to organize the presentation layer [online]. [cit. 2018-04-06]. Dostupné z: https://antonioleiva.com/mvp-android/

[29] MVVM architecture, ViewModel and LiveData (Part 1) [online]. [cit. 2018-04-06]. Dostupné z: https://proandroiddev.com/mvvm-architecture-viewmodel-and-livedata-part-1-604f50cda1

[30] Introduction to Android Architecture Components [online]. [cit. 2018-04-06]. Dostupné z: https://android.jlelse.eu/introduction-to-android-architecture-components-22b8c84f0b9d

SEZNAM SYMBOLŮ, VELIČIN A ZKRATEK

OS	Operational Systém, Operační Systém
UI	User Interface, uživatelské rozhraní.
IDE	Integrated Development Environment
DEX	Dalvik Executable
APK	Android Application Package
OHA	Open Handset Alliance
JIT	Just-In-Time Compiler
AOT	Ahead-of-time Compiler
ART	Android RunTime
NFC	Near field communication
HAL	Hardware Abstraction Layer
API	Application Programming Interface
JSON	JavaScript Object Notation
XML	Extensible Markup Language
MVC	Model-View-Presenter
MVVM	Model-View-ViewModel